University of Stellenbosch

Department of Civil Engineering

Object-Oriented Finite Element Framework

AH Olivier

12190462

Thesis presented in partial fulfilment of
the requirements for the degree of
Masters of Civil Engineering at the
University of Stellenbosch.

Study leader: Dr GC van Rooyen

December 2002

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.

Ek, die ondergetekende verklaar hiermee dat die werk gedoen in hierdie tesis my eie oorspronklike werk is wat nog nie voorheen gedeeltelik of volledig by enige universiteit vir 'n graad aangebied is nie.

Signature: Date:

Synopsis

The role of the computer has changed from a calculation tool to a tool that supports human thinking.

In this thesis fundamental aspects of the Finite Element method are mapped to an object model with a well defined structure which provides for local and distributed analysis work.

To achieve this the following was investigated:

- An object-oriented framework for the Finite Element analysis method
- An associated graphical user interface that enables the user to create and modify Finite Element models in an effective way
- Requirements for the sharing of analysis information in a communication network

Proposed solutions are implemented in a pilot application which indicates their potential.

Opsomming

Die rol van die rekenaar het verander vanaf 'n gereedskapstuk wat berekening doen na 'n gereedskapstuk wat menslike denke ondersteun.

In hierdie tesis word die fundamentele aspekte van die Eindige Element metode oorgedra na 'n objek model met 'n goed gedefinieerde struktuur wat lokale en verspreide analisering werk ondersteun.

Om dit te bereik is die volgende ondesoek:

- 'n Objek orienteerde raamwerk vir die Eindige Element metode
- 'n Geassosieerde grafiese raamwerk wat die gebruiker in staat stel om objekte te skep en te verander
- Vereistes vir die deel van analise inligting in 'n kommunikasie netwerk

Die voorgestelde oplossing is geimplimenteer in 'n loodsimplementering wat die voordele van die benadering uitlig.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

Glossary

dof(s)	degree(s) of freedom
pid	persistent identifier
sid	selection identifier
CST element	constant strain triangular element
GUI	Graphical User Interface

1. Problem statement

The goal of this study is three fold:

1. To develop and implement an object-oriented framework for Finite Element analysis.
2. To develop an associated framework for graphical user interaction.
3. To develop facilities for deploying the above frameworks in a distributed communication network.

1.1 Object-oriented framework for Finite Element analysis

This part of the thesis defines an object-oriented framework for the Finite Element method. The necessary interfaces and classes are developed and implemented.

Objective:

A successful framework would have a modular structure, which can easily be extended. To implement a new element for example, there should be no need to rewrite existing classes. Instead, classes required to define the new element should implement well-defined interfaces and extend given abstract classes.

1.2 Associated framework for graphical user interaction

This part of the thesis defines an object-oriented framework for graphical user interaction, which complements the Finite Element framework. The necessary interfaces and classes are developed and implemented.

Objectives:

The user interface must be easy to use and minimize the amount of work to be done by the user.

Furthermore, the user interface must be flexible so that new graphical elements can be developed by implementing well-defined interfaces.

The user should be able to obtain specific information about specific, i.e. selected objects of the model.

1.3 Deploying the frameworks in a distributed communication network

The final part of the thesis is aimed at supporting multiple users in collaborating in the execution of a project's analysis tasks. This requires facilities for the distribution and exchange of the project's analysis parameters.

Objectives:

A successful implementation in a distributed environment will allow the users to work efficiently, while collaborating as much as possible. Due to the nature of structural analysis, a standalone capability is also required, which creates problems in keeping information consistent. The user must be in control of the updating of the information base. The proposed solution must be media suitable i.e. well supported by existing technologies. Furthermore, the user needs tools which are useful in the distributed environment. In this thesis, only the fundamental requirements in support of collaboration are introduced.

2. Brief Background of Structural Analysis

2.1 Structural Engineering

Structural engineering centres about the conception, design, and construction of structural systems needed in support of human activities. Structural analysis is an integrated process of the structural design cycle. Figure 1 shows the engineering design cycle and the role that structural analysis plays within the cycle.

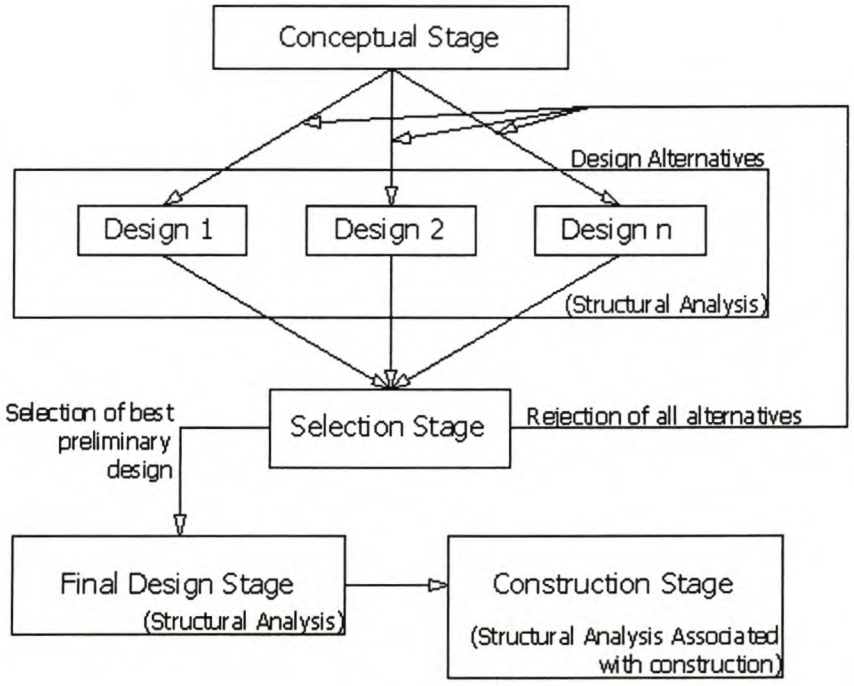


Figure 1 The engineering design process. [2]

2.2 Concept of structural analysis

Structural analysis is the process whereby the physical behaviour of a structure is predicted. The behaviour (i.e. response) of the structure is a measure of how the structure reacts to given external influences. The laws of physics describe the actual relation between influences on a structure and its physical response. The brief of structural analysis is to mathematically describe the relation between influences and response for structural components, and to find a solution to the equations at hand.

Several handbooks explaining different analysis methods are available. [1,3,7,8]

2.3 Finite Element method

In this thesis, the Finite Element method is considered to be the de facto standard in the analysis of structures. The Finite Element method is based on the subdivision of the problem domain into a mesh of finite elements. The shape is described by the topology of the elements, the geometry of their nodes and interpolation of the geometry inside and along the edges of the elements. The physical state is interpolated in terms of generalised variables (the degrees of freedom of the Finite Element analysis) at chosen state points (usually at the nodes). In structural mechanics the displacement method is popular, and the degrees of freedom are displacement and/or rotation components.

The integral equation, describing behaviour over the problem domain, is expressed as the sum of integrals over the elements. Substitution of the interpolation functions into element integrals yields expressions of element matrices and vectors. The integrals are computed analytically, if possible, otherwise numerically. Summation of the element contributions yields, for the Finite Element method, a system of algebraic equations of the form $[\mathbf{K}]\{\mathbf{U}\}=\{\mathbf{Q}\}$. Given an appropriate set of boundary conditions, the unknown variables at the state points are determined by solving the algebraic equations. Once all the variables at the state points are known, the physical state inside each element can be computed. This represents a Finite Element solution of the problem.

The Finite Element method's versatility and robustness are its outstanding features: The size and shape of elements are optional, and elements of different types and physical properties can be mixed, arbitrary geometries can be used and loads and boundary conditions can be modified. Furthermore, the Finite Element mesh is a mathematical abstraction that is easy to visualize. As a result, the Finite Element method has become the standard in the solution of structural mechanics problems.

2.3.1 Components of Finite Element method

The following components can be identified when examining the Finite Element method.

2.3.1.1 Degrees of freedom (dofs)

The degrees of freedom are the generalized variables at the chosen state points from which the physical state inside the element can be interpolated. In structural analysis the displacement method is popular and displacements/rotations are chosen as degrees of freedom. Any three-dimensional displacement of a point can be described by three linearly independent translations and three linearly independent rotations.

2.3.1.2 Nodes

Nodes are the points that define the finite element mesh used to interpolate the geometry of the problem domain. For practical reasons, the state points are chosen to coincide with the nodes, although it is not compulsory to do so.

2.3.1.3 Elements

Both geometry and physical behaviour are interpolated inside elements using suitable interpolation functions. Substitution of the interpolation functions into element integrals which describe the structural behaviour yield element matrices and vectors, called stiffness matrices and load vectors (see 2.3.1.5) in structural problems. Summation of the element contributions yields the stiffness matrix and load vector of the system.

2.3.1.4 Supports

Supports are the points on the boundary where kinematic conditions of state are prescribed i.e. the displacements are prescribed, and the reaction forces are unknown.

2.3.1.5 Loads

Loads describe the static conditions of state on the boundary, e.g. externally applied pressures, forces and moments. Loads cause the structure to deform and internal forces to develop.

2.3.1.6 Constraints

Linear constraints define linear relationships between master and slave degrees of freedom. As a result the slave dofs can be eliminated from the system of unknowns. Constraints are useful when different elements are combined.

2.3.1.7 Local coordinate systems

Local coordinate systems allow the description of boundary conditions and results in terms of specific coordinate systems, which do not coincide with the global coordinate system.

2.4 The new role of the computer in structural analysis

The development of the computer industry caused a fundamental change in the analysis of structures. Today it is not only possible to solve a large number of equations in a fraction of a second. With the development of object-oriented languages such as Java and C++ the role of the computer changes from a calculation apparatus to a tool that supports human thinking.

The object-oriented approach allows the engineer to map a mathematical abstraction of a structure to the computer.

For example:

- In a procedural environment, a truss will be represented by a table of nodes, a table of elements, a table of supports, a table with material properties, a table with cross sectional properties and a table with loads. The data within a table only has meaning when given to a procedure that interprets the table. Outside the context of the procedure, the data is meaningless. The data is simple and the procedures are clever.
- In an object-oriented environment, a truss will be represented by a set of nodes, a set of elements where each element has a material and cross section, a set of supports and a set of loads. An element, for example, will have certain functionalities, like being able to calculate it's own mass and length. The data becomes clever and the procedures simple. In the next chapter, the object-oriented approach will be discussed in more detail.

3. Object-Oriented Models for Structural Analysis

A model maps mental concepts regarding a system of the real world to the computer. The object-oriented approach is based on the premise that real world systems consist of separable entities that have certain properties and functionalities. The entities are mapped to a structured set of software objects called an object model. The structure of the object model is laid down in the relations on the objects. The class structure of the object model is the central concept in object-orientation, and is the mechanism for implementing equivalence relations in object models.

3.1 Identification of objects

The identification of objects is crucial for the successful implementation of the model concept. Objects generally refer to other objects in the memory space of a computer. These references are time depended. *Persistent identifiers* (pids) are introduced to enable the objects to establish references to one another.

3.1.1 Persistent identification

The identifier of an object is called persistent if it is independent of the current location of the object in memory or in a file. *Strings* are suitable persistent identifiers. They correspond to the names of persons.

3.1.2 Temporary identification

Objects are temporarily identified by their addresses in memory or in a file. A compiler maps declared variable names to temporary identifiers, which are lost at the end of a session.

Persistent identifiers are essential to establish the relationships between objects. If an object knows the persistent identifier of another object with which it has a relationship, it is possible to find the temporary identifier (i.e. address space in the memory of the computer) of that object. A map is used to link persistent identifiers of objects to the references of the objects. This mechanism enables objects to obtain the references of related objects via their *persistent identifiers*.

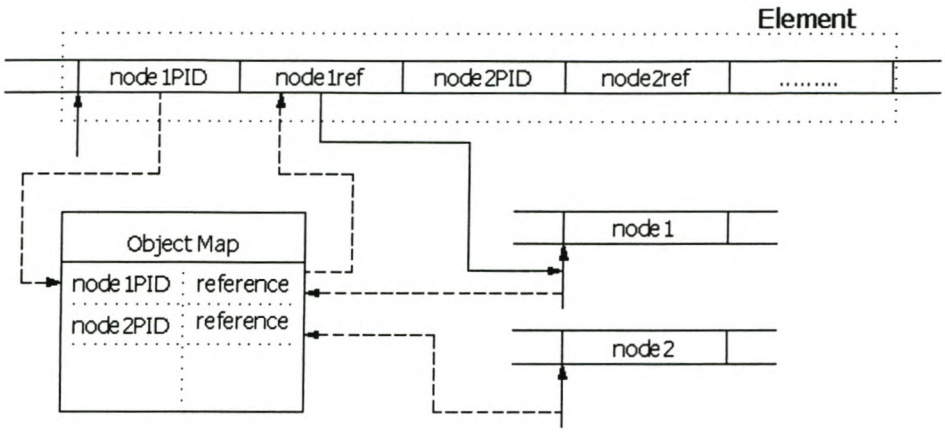


Figure 2 The setting of references

Figure 2 shows the process of establishing the references once all the objects are instantiated. The constructor of each object registers the object in the object map. The key used for this reference is the persistent identifier of the object. The reference in this map is the location of the object in the memory space of the computer.

To establish the references to a related object, the temporary identifier (reference) is obtained from the object map and stored in the object. This enables the object to access the related object directly.

4. Interfaces

Interfaces are used to define the required functionality of objects in the Finite Element framework. By implementing the interfaces, the various classes provide the functionality defined in the interfaces. A class represents an abstraction of a real object e.g. an element.

Advantages of using interfaces: Instances of all classes that implement a specific interface are equivalent at the functional level defined by the interface. An interface is consequently a mechanism for establishing an equivalence relation in an object model, which has the advantage that the elements of the equivalent relation can be dealt with in an identical way. For example, when using different types of finite elements, all the elements that implement the `IElement` interface will have the ability to provide an element stiffness matrix when asked for it by the `analysis` object. For the analysis object, there is no difference between a Constant strain triangular (CST) element and a truss element. It simply obtains the element stiffness matrix of each element and assembles the system stiffness matrix in order to perform the analysis. As a result, new elements can be developed without changing existing code. To develop a new element, the developer only needs to implement the `IElement` interface, and extend the abstract `Element` class. The existing application will be able to use the new element without any changes.

4.1 Interface Hierarchy

Figure 3 shows the interface hierarchy that was developed for the Finite Element method. The remainder or this chapter will briefly describe each interface. A more detailed description is provided in Appendix A.

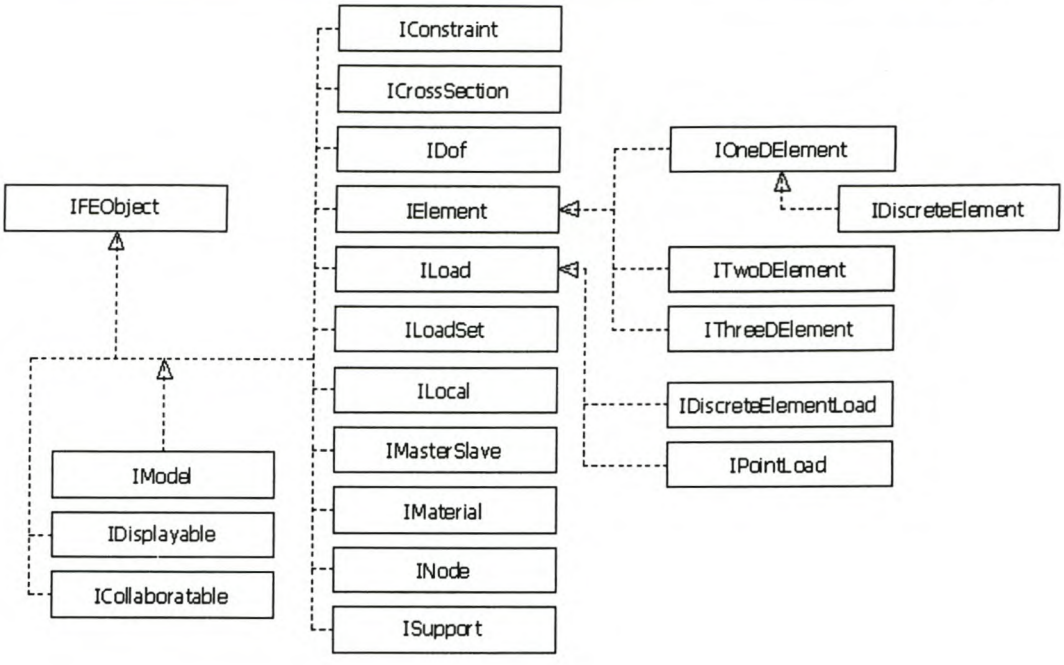


Figure 3 The interface hierarchy.

4.2 Interface descriptions

4.2.1 IFEObject

This interface prescribes the functionality that is needed in all the finite element components. It defines only one method:

- public *String* **getPID()**

This method will ensure that all the finite element components will have the ability to return a *persistent identifier* when asked for it.

4.2.2 IDisplayable

This interface prescribes the functionality that the graphical user interface requires from objects that are displayed by it. It allows the graphical user interface to select objects by clicking on them (mouse input) and to manipulate the object graphically. Important methods in this interface is:

- public *void* **draw(** *IGuiPanel gp* **)**

  This method enables an object to draw itself on an `IGuiPanel` that is handed to the object.

- public *boolean* **containsPoint(** *Point p* **)**

  This method receives a point in the model coordinate system, and will return true if the point is located on/inside the graphical (i.e. the physical) boundary of the object, and false otherwise. Thus, the object has the ability to determine whether it has been selected.

4.2.3 ICollaboratable

This interface prescribes the functionality needed by an object if it is to be used in a distributed environment by multiple users.

Important methods prescribed by this interface:

- public *void* **getSID( )**

  This method returns the *selection identifier* of the object. The selection identifier is used to accommodate different versions of the object. While the object is inside a runtime model, the *persistent identifier* is used to establish references. Outside the runtime model, an object may have different versions. Then it is important to distinguish between versions, thus the *selection identifier* is introduced to make selection possible. *Note:* only one version of an object can be used inside a model at a time.

- public `Version` **getVersion( )**

  This method returns the version object of an object. The version object keeps track of the version of the object and contains information like the version number, the user responsible for the creation of this version, etc. See appendix A for a detailed description of this object.

- public *void* **updateFundamentalChange(** *UpdateGene updateGene* **)**

  This method enables an object to modify itself in order to keep it consistent with changes that occurred elsewhere. The `UpdateGene` contains the information required to perform an update, e.g. the method that caused the change and the new attribute values.

4.2.4 INode

This interface prescribes the functionality that nodes of the finite element mesh must have.

Important methods of this interface are:

- public *void* **setActiveDOFs(** *IModel model, boolean[ ] activeDOF* **)**

  This method activates the required degrees of freedom at a node (if they are not already active). The model parameter sets the model context within which the node is being used.

- public *boolean[ ]* **getActiveDOFs( )**

  This method returns a boolean array with the same number of indices as the total number of degrees of freedom at a node (active and inactive). A true value will indicate that the degree of freedom is active and exists, and a false value will indicate that the degree of freedom is not required by the set of elements connected to the node.

- public *int[ ]* **getSystemIndices( )**

  This method returns the system equation indices of the node's degrees of freedom.

4.2.5 IElement

This interface prescribes the functionality that elements using this framework for Finite Element analysis must provide.

Important methods are:

- public *double[ ][ ]* **getStiffnessMatrix( )**

  This method returns the element stiffness matrix in the global coordinate system.

- public *int[ ]* **getSystemIndices( )**

  This method returns the system equation numbers of the element degrees of freedom, which enables the **analysis** object to assemble the system stiffness matrix correctly.

- public *double[ ][ ]* **getRotationMatrix( )**

  This method returns the matrix that transforms displacements in the global coordinate system to displacements in the element's local coordinate system. This is an orthogonal matrix. The transpose matrix of an orthogonal matrix is

equal to the inverse of the matrix. Thus, the transposed rotation matrix will transform displacements in the local coordinate system to displacements in the global coordinate system.

- public *double[ ]* **getElementResultVector( )**

  This method returns a vector with the element's field variable(s) resulting from the analysis.

4.2.5.1 IOneDElement

A 1D element is an element that has a constant cross section along the axis of the element, e.g. a prismatic beam element.

Important methods are:

- public *double* **getRotationAngle( )**

  This method returns the angle between the vertical plane that contains the *x-axis* and the element *xy-plane* (i.e. the plane that contains both the element *x-axis* and the element *y-axis*.) (Right hand rotation about local x-axis)

- public *double[ ] [ ]* **getElementDirectionVectors( )**

  This method returns the element direction vectors (i.e. the element coordinate system). The local element coordinate system is taken as follows (See Figure 9):

  x-axis in the direction of the element. (From the first node to second node)

  y-axis in the vertical plane that contains the x-axis (if the rotation angle is equal to zero)

  z-axis using the right hand rule to determine the direction.

  If the *x-axis* is vertical and the rotation angle is zero, the local *y-axis* is taken in the direction of the global Z-axis.

- public *double* **getLength( )**

  This method returns the length of the element.

4.2.5.2 IDiscreteElement

This interface defines the functionality of discrete elements. Discrete elements are elements like the standard Euler beam element. There is no need to subdivide a discrete element into smaller elements to improve the accuracy of the finite element solution. A discrete element calculates the fixed-end forces due the internal loads. When internal forces are computed, the fixed-end forces are subtracted from the internal forces due to nodal displacements to give an analytically exact solution.

Important methods are:

- public *void* **addLoad(** *IDiscreteElementLoad load* **)**

  This method adds a discrete element load to a discrete element.

- public *void* **incrementFefs(** *String key, double[ ] values* **)**

  This method increments the fix-end-force vector with the fixed-end forces passed to it.

4.2.5.3 ITwoDElement

This interface defines the functionality of two-dimensional elements. A two-dimensional element is a plate or membrane element that is defined in two dimensions.

Important methods are:

- public *double[ ]* **normalDirectionVector( )**

  This method returns the unit vector that is normal to the plane that contains the element.

- public *double* **getArea( )**

  This method returns the area of the element.

- public *double* **getThickness( )**

  This method returns the thickness of the element.

4.2.5.4 IThreeDElement

This interface defines the functionality of three-dimensional elements. Three-dimensional elements are not implemented in the pilot application.

Important method is:

- public *double* **getVolume( )**

  This method returns the volume of the element.

4.2.6 ILoad

A load is a vector that has an intensity and a direction. By separating the direction and the intensity, the user can enter a direction vector that is convenient, e.g. {+1, -3, +4} and an intensity.

Important methods are:

- public *double[ ]* **getLoadVector( )**

  This method returns the load components in the global system directions (i.e. global coordinates of the load vector)

- public *double[ ]* **getDirectionVector( )**

  This method returns the direction vector of the load.

- public *int[ ]* **getSystemIndices( )**

  This method returns the system indices that are assigned to the load vector components.

4.2.6.1 IDiscreteElementLoad

Interface `IDiscreteElementLoad` defines the functionality of *discrete element loads*. Discrete element loads are loads that act on discrete elements only. A discrete element load can compute its contribution to the internal force distribution in a discrete element. With discrete elements, exact internal force results are calculated, not depending on the size of the elements.

Important methods are:

- public *double[ ]* **getAFDistribution(***IDiscreteElement element, double[ ] afd* **)**

  This method calculates the axial force distribution caused by the discrete element load and adds it to the axial force distribution of the element.

- public *double[ ]* **getSFDistribution(***IDiscreteElement element, double[ ] sfd* **)**

  This method calculates the shear force distribution caused by the discrete element load and adds it to the shear force distribution of the element.

- public *double[ ]* **getBMDistribution(***IDiscreteElement element, double[ ] bmd***)**

  This method calculates the bending moment distribution caused by the discrete element load and adds it to the bending moment distribution of the element.

4.2.6.2 IPointLoad

This interface defines the functionality of concentrated loads that are not applied at a node.

Important method:

- public *double[ ]* **getOffSet( )**

  This method returns the offsets of the point load from the element nodes. In the one-dimensional case, an offset is equal to the distance from the first node to the point load divided by the distance between the first and second node of the element.

4.2.7 ISupport

This interface defines the functionality of the `Supports`. In the pilot application, there are six possible support conditions: three translations and three rotations. The active degrees of freedom are determined by the node where the support exists. The support conditions only specify that if the nodal degree of freedom is active, then the support condition apply. For example, if the Z-translation is not active (determined by the elements that connects to the node), it is not part of the system of equations, and by default, no support is needed to support the dof even if the supportDOF indicates that the user requires a support.

Important methods are:

- public *int* **getNumberOfSupportDOFs( )**

  This method returns the number of active support degrees of freedom.

- public *double[ ]* **getPrescribedDisplacements( )**

  This method returns the prescribed displacements of the support.

- public *double[ ]* **getReactions( )**

  This method returns the reactions at the support.

- public *boolean[ ]* **getSupportConditions( )**

  This method returns an array stating whether a support condition is active (true) of inactive (false).

4.2.8 IConstraint

Interface *IConstraint* defines the basic functionality of `Constraint` objects. A constraint object generates *MasterSlave* objects during the analysis of a model.

Important methods are:

- public *void* **addDOFEquation**(*int masterDOFindex, int slaveDOFindex, String value*)

  This method adds a degree of freedom equation to the constraint object. (See 5.5.4)

- public *void* **createMasterSlave(** *IModel model* **)**

  During the analysis of a model, this method creates the `MasterSlave` objects (represented by the `Constraint`) and adds it to the model.

4.2.8.1 IMasterSlave

Interface `IMasterSlave` defines the basic functionality of `MasterSlave` objects. A `MasterSlave` object is an object that contains a relationship between a master degree of freedom and a slave degree of freedom. `MasterSlave` objects are transient. They are created by the `Constraints` during the analysis of the model, and are of little direct use for the user.

Important methods are:

- public *IDof* **getMasterDof( )**

  This method returns the mater degree of freedom of this `MasterSlave`.

- public *IDof* **getSlaveDof( )**

  This method returns the slave degree of freedom of this `MasterSlave`.

- public *double* **getValue( )**

  This method returns a constant value that describes the slave dof. as a function of the master dof.

4.2.9 ICrossSection

Interface `ICrossSection` defines the functionality of cross sectional properties for prismatic one-dimensional elements. Figure 4 shows the local coordinate system of a cross section. (Note: This coordinate system is different from the element's local coordinate system)

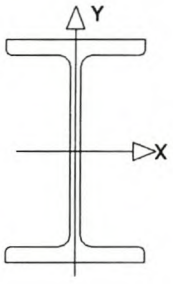


Figure 4 Local coordinate system of a cross section

Important methods:

- public *double* **getArea( )**

  This method returns the area of the `CrossSection`.

- public *double* **getIxx( )**

  This method returns the second moment of area of the `CrossSection` about the x-x axis.

- public *double* **getIyy( )**

  This method returns the second moment of area of the `CrossSection` about the y-y axis.

- public *double* **getJ( )**

  This method returns the St Venant torsion property of the `CrossSection`.

4.2.10 IMaterial

This interface defines the functionality of materials used in this framework.

Important methods:

- public *double* **getEModulus( )**

  This method returns the Young's modulus of the material.

- public *double* **getPoissonRatio( )**

  This method returns the Poisson ratio of the material.

4.2.11 IDof

Interface `IDof` defines the basic functionality of an object representing a degree of freedom, i.e. instances of class `Dof`. Degrees of freedom (dofs) are the state variables of the system.

States of `Dof` objects:

A degree of freedom is either a primal or a dual degree of freedom.

A *primal* dof is a dof where the primal value (i.e. the displacement or rotation in the case of displacement-based elements) is unknown. It can be divided into two catagories:

- Master dofs – the primal value is determined during the solution of the system equations.
- Slave dofs – the primal value is a function of other primal values in the system.

Dual dofs are degrees of freedom on the boundary that are prescribed (i.e. the displacements are known and the reactions are unkown).

Note: it is not possible for a slave dof to be a dual degree of freedom because the displacement of a dual dof is known by definition.

Important methods:

- public *double* **getPrimalValue( )**

  This method returns the primal value (i.e. the displacement).

- public *double* **getDualValue( )**

  This method returns the dual value of the degree of freedom.

- public *int* **getIndex( )**

  This method returns the equation number of the degree of freedom.

- public *boolean* **isMaster( )**

  This method returns true if the degree of freedom is a master dof.

- public *boolean* **isPrimal( )**

  This method returns true if the primal value of the dof. is unknown, false otherwise.

4.2.12 ILocal

Interface `ILocal` describes the basic functionality of a local coordinate system. A local coordinate system is completely defined by the direction of its *x-axis*, and the rotation of the system about that axis. For a rotation angle of 0°, the local xy plane is vertical. If the local *x-axis* is vertical, the local *y-axis* is taken in the global Z-direction.

Important method:

- public *void* **setDirectionVectors(** *double[ ] direction 1, double angle)*

  This method computes the three direction vectors that represent the local coordinate system. Direction 1 is the direction of the local *x-axis*. The angle parameter allows the user to rotate the *y- and z-axis* about the *x-axis*. For an angle of 0°, the local *y-axis* is in the vertical plane that contains the *x-axis*.

4.2.13 ILoadSet

A load set is a set of loads that act together on a model. When the system equations are solved, the result vector is saved in the load set.

Important methods:

- public *void* **add (** *ILoad load* **)**

  This method adds an `ILoad` to the load set.

- public *void* **remove(** *ILoad load* **)**

  This method removes an `ILoad` from the load set.

- public *double[ ]* **getLoadVector(** *int dimension, IModel model)*

  This method returns the load vector of the load set.

- public *double[ ]* **getResultVector( )**

  This method returns the result vector, i.e. a vector that contains the values that were computed during the analysis of the model.

4.2.14 IModel

The `IModel` interface describes the functionality of `Model` objects. A `Model` object encapsulates all components needed for the analysis of a structure using the finite element method.

Important methods:

- public *boolean* **addComponent(** *FEObject component* **)**

  This method adds an `FEObject` to the model.

- public *boolean* **addSet(** *String interfaceName* **)**

  This method adds a new component set to the model.

- public *void* **analyse( )**

  This method performs the analysis of the model.

- public *void* **setResults(** *ILoadSet loadset***)**

  This method sets the computed values of the `Dof` objects. These values are calculated during the analysis of the model and saved in the resultVector of the `LoadSet` objects.

5. Pilot application

This chapter describes the implementation of the interfaces in the pilot application. The interaction between various objects of the different classes is discussed. The pilot application is limited to linear elastic analysis with small displacements and linear material models. The data structures of the application and the model are described as well.

5.1 The Finite Element Application

Figure 5 shows the basic data structure of the Finite Element application. The static class `FEApplication` contains a HashMap that maps all the `FEObjects` in the memory. The *selection identifiers* of the `FEObjects` are used as keys to the `FEObjects` in the application map. A *selection identifier* uniquely identifies a specific version of an object. This is done by combining the *persistent identifier* with the version number. By using *selection identifiers* to reference objects in the application map, different versions of the same object can exist in the application space. This is necessary since multiple models are allowed for.

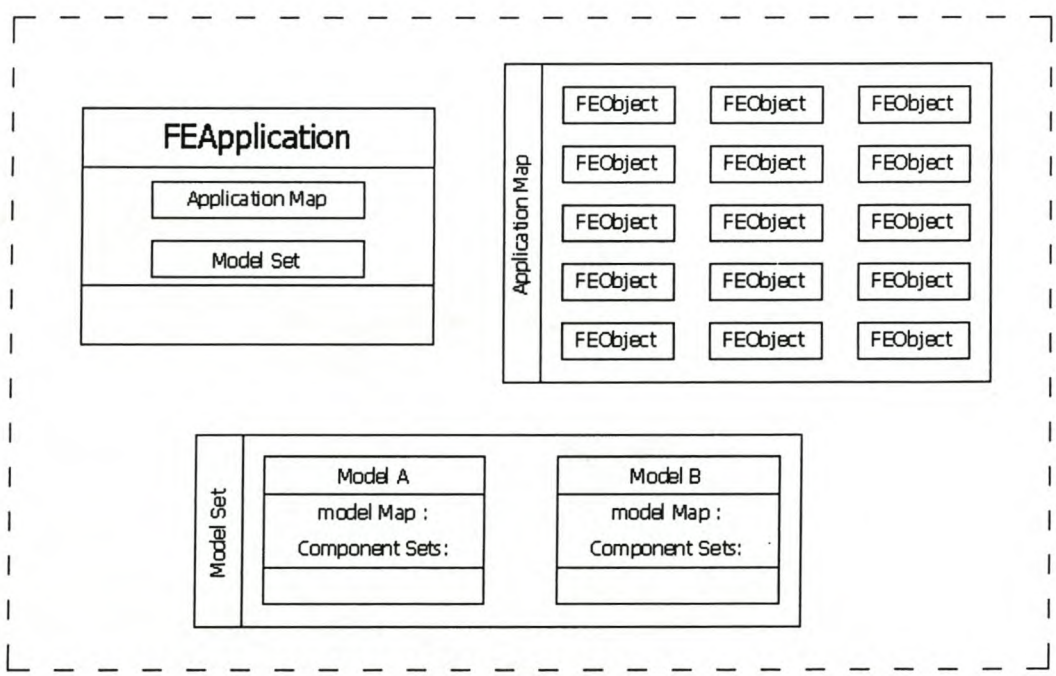


Figure 5 The application data structure.

5.2 The model

Each model object contains two HashMaps.

- The first HashMap is the model map. All the `FEObjects` that are part of the model are referenced in this map. The keys that are used for this reference are the *persistent identifiers* of the `FEObjects`. Thus, by knowing the *persistent identifier* of an `FEObject`, the object is easily found.
- The second HashMap stores the different component sets. In this case, the key associated with a component set is a String that equals the name of the implemented interface of the component set.
    - A component set is a set of objects that implement the same interface. This allows for operations on functionally equivalent objects in a model. If an `FEObject` implements an interface and there exists a component set with a key that matches the interface name, the `FEObject` will be referenced in that component set.

Figure 6 shows a model with a model map and component sets. The four `FEObjects` are all referenced in the model map (they belong to this model). They are also referenced in the component sets whose keys match the interface names that they implement.

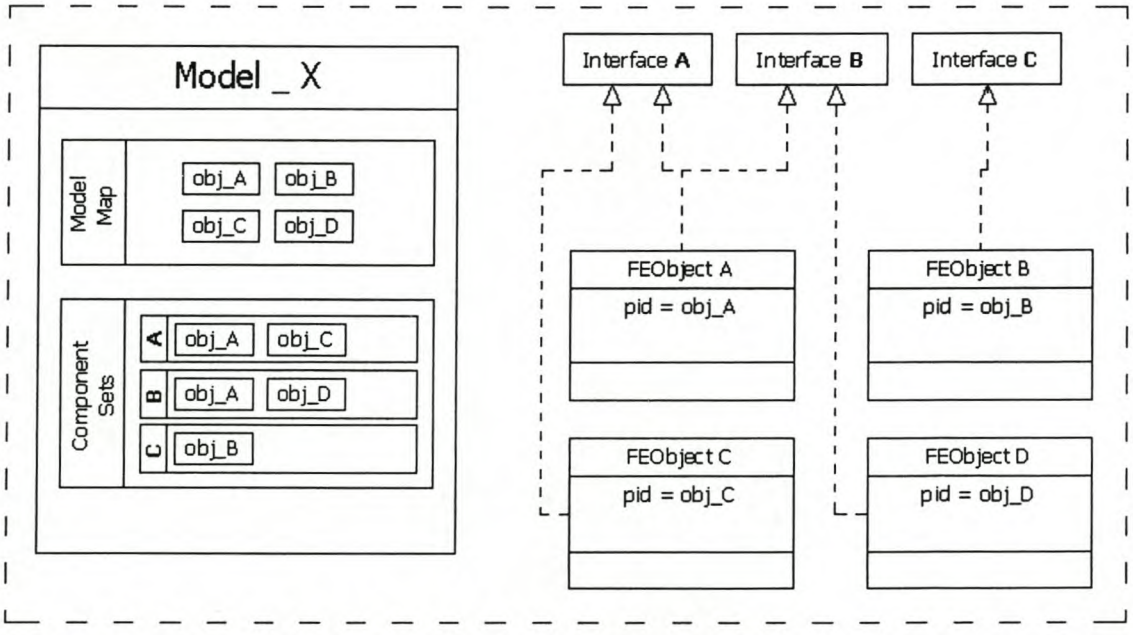


Figure 6 The model data structure

5.3 The FEObject

All the Finite Element components extend the abstract class `FEObject`. The constructor of this class creates the first version and adds the new object to the *application map*.

5.3.1 Steps in creating a new FEObject

- The Graphical User Interface (GUI) gets the information from the user, e.g. the two nodes that form the start point and end point of a *frame element*.
- A *persistent identifier* is obtained from the *persistent identifier service*.
- The constructor of the new component is called. The *persistent identifier* and the other fundamental attributes are handed to it. A fundamental attribute is an attribute that defines the object, and is not time or location dependent.
- The constructor calls the *super class* constructor, and passes the *pid* to it. This step repeats until the constructor of `FEObject` is called.
- The `FEObject` constructor creates the first version attribute of this object, and references the `FEObject` in the *application map*. (The object's *selection identifier* is used for this mapping)
- In the GUI, the method *setReferences( )* of the new object is invoked and the object is passed to the method *addComponent(FEObject obj)* in the active model. (The invocation of the *setReferences( )* method enables the `FEObject` to graphically represent itself)
- The active model will add the object to the model map (using the *pid* as key value), and reference it in all the component sets where the key of the component set matches an interface name that is implemented by the object handed to the model.

The fundamental attributes of an `FEObject` are the following:

- The *persistent identifier* of the object
- The *version* of the object

5.4 The global coordinate system

The pilot application uses a right hand global coordinate system with the Y-axis vertical.

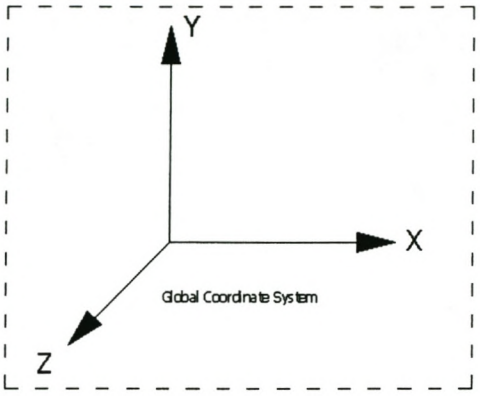


Figure 7 Global Coordinate System

The global direction vectors are the following:

- X direction = [1; 0; 0]
- Y direction = [0; 1; 0]
- Z direction = [0; 0; 1]

5.5 The Finite Element components

The remainder of this chapter describes all the finite element components in more detail.

5.5.1 Class Node

This class implements the following interfaces: `INode`, `IDisplayable`, `ICollaboratable`, `IFEObject`.

Nodes are the points that define the finite element mesh used to interpolate the geometry of the problem domain. For convenience, the state variables are also located at the nodes. In the pilot application, displacement-based elements are used, thus, displacements and rotations are calculated at the nodes.

Nodes may have a total of six degrees of freedom (three translations and three rotations). Each node has an `IDof` array of size six. If, for example, degree of freedom 2 is active at the node, a `Dof` object will be created, which is referenced in the second cell of the `IDof` array. The boolean array, activeDOF, keeps track of the status of the degrees of freedom.

The fundamental attributes are the following:

- The *coordinates* of the node
- The *persistent identifier* of the local coordinate system, if applicable

5.5.2 Class Local

Class Local extends class `FEObject` and implements the following interfaces: `ILocal, IDisplayable, ICollaboratable.`

Class `Local` allows the user to define a coordinate system at a node other than the global coordinate system. A local coordinate system is completely defined by the direction of the local x-axis, and the rotation of the system about the x-axis. For a rotation angle of 0°, the local xy plane is vertical. If the local x-axis is vertical, the local y-axis is taken in the global Z-direction.

The rotation angle rotates the local xy plane about the local x-axis, using the right hand rule, if the thumb is pointing in the direction of the local x-axis, the fingers will point in the positive direction of the angle.

The direction vectors of a `Local` instance define the directions of the local coordinate system.

The fundamental attributes are the following:

- The *persistent identifier* of the node where the `Local` exists
- The first direction vector
- The rotation angle

5.5.3 Class Dof

Class `Dof` extends class `FEObject` and implements the following interface: `IDof`.

Degree of freedom objects are created by the nodes during the invocation of the *setActiveDOF( )* method of `node` objects. They are time dependent and not persistent. Thus, a degree of freedom object has no fundamental attributes. In paragraph 4.2.11 the state of a degree of freedom is discussed.

5.5.4 Class Constraint

Class `Constraint` extends class `FEObject` and implements the following interfaces: `IConstraint, IDisplayable, ICollaboratable, IFEObject`.

This class allows the user to describe the displacements of a node in terms of the displacements of another node. A `Constraint` object contains one or more `MasterSlave` objects.

The fundamental attributes are the following:

- The *persistent identifier* of the master node
- The *persistent identifier* of the slave node
- Dof indices at the master node
- Dof indices at the slave node
- Values that define the relations between the master and slave indices

5.5.4.1 Class MasterSlave

Class `MasterSlave` extends class `FEObject`. Instances of this class are time and location dependent. Objects of this type are created during the analysis of the structure. The information required to create the set of `MasterSlave` objects are contained in the set of constraint objects. Each `MasterSlave` object contains one element of the system compatibility matrix, **[*A*]**.

$$d^* = [A]\, d \qquad \text{....1}$$

d^* = all the dofs

d = master dofs

5.5.4.1.1 The compatibility matrix

The compatibility matrix is used to remove the slave degrees of freedom from the system of equations. The number of rows in this matrix is equal to the number of degrees of freedom of the complete system. The number of columns of this matrix is equal to the number of master degrees of freedom. The i^{th} row of the matrix contains the coefficients relating the i^{th} degree of freedom to the master degrees of freedom. The [**A**] matrix is sparsely populated. The number of entries in this matrix is equal to the number of master degrees of freedom plus the number of `MasterSlave` objects. See paragraph 9.2 for the implementation of the sparse matrix.

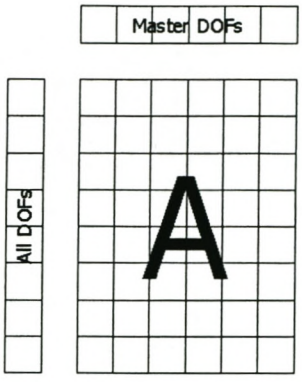


Figure 8 The compatibility matrix

Each `MasterSlave` object contains a value representing an entry of the **[A]** matrix. The equation below yields the condensed system matrix.

$$[K_s] = [A]^T [K_s^*] [A] \qquad \text{....2}$$

$[K_s^*]$ = Complete system matrix

$[K_s]$ = Condensed system matrix

5.5.5 Class Element

Class `Element` is an abstract class that extends class `FEObject` and implements the following interfaces: `IElement, IFEObject`.

The fundamental attributes are the following:

- The *persistent identifiers* of the nodes that define the element
- The *persistent identifier* of the material object of this element

Four elements were extended from this class for the purpose of the pilot application.

In 1D elements, the element direction vectors are defined by the following set of rules.

- The first element direction vector is parallel to the element, from the first node to the second node.
- The second element direction vector is located in the vertical plane that contains the first element direction vector.
- The third element direction vector is calculated as the cross product of the first and the second element direction vectors.
- **If** the first element direction vector is vertical, the second element direction vector is taken in the global Z direction.

1D beam elements may be rotated about the element's *x-axis*, in which case the element's *y-axis* will lie in a plane which is not vertical any longer.

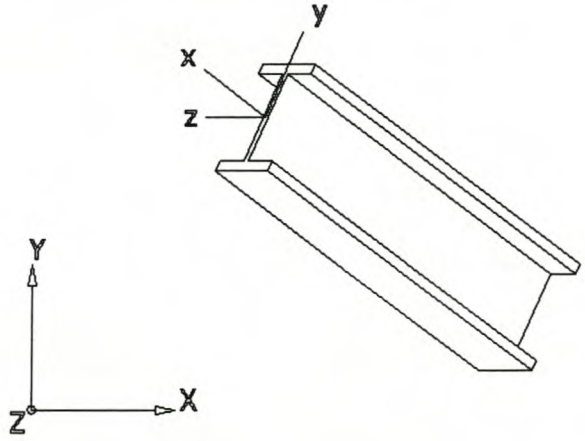


Figure 9 Local element directions

In 2D elements, the element direction vector is a single vector normal on the plane of the element.

5.5.5.1 Local and global coordinate systems

Class `Element` has an abstract method *getStiffnessMatrix( )*. This method is implemented by all the elements that extend class `Element`. Firstly, an element calculates the local element stiffness matrix. The local element stiffness matrix must be transformed to an equivalent stiffness matrix in the global coordinate system. This enables the assembly of $[\mathbf{K_s}]$, the system stiffness matrix.

This transformation is done by the following equation:

$$[K_{eG}] = [A_{rot}]^T [K_{eL}] [A_{rot}] \qquad \text{....3}$$

$[A_{rot}]$ = rotation matrix

$[K_{eG}]$ = K_e in global coordinate system

$[K_{eL}]$ = K_e in local element coordinate system

Matrix $[\mathbf{A_{rot}}]$ is the matrix that transforms displacements in the global coordinate system to displacements in the local element coordinate system. Figure 10 shows the transformation from the global coordinate system to the local coordinate system in two dimensions. Computing the dot product between two direction vectors, the projection of the one direction vector in the direction of the other direction vector is obtained.

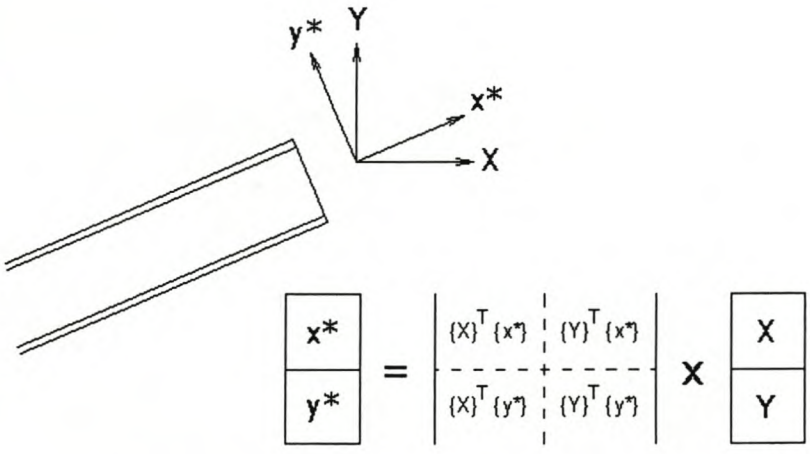


Figure 10 Transformation of local element coordinates

5.5.5.2 2D space elements in a 3D model

A 2D space element is an element that is defined and used in a two-dimensional plane only. Two-dimensional space elements are usually used in 2D analysis applications. In this thesis, 2D space elements are used in a 3D space environment. See paragraph 9.1 for a detailed discussion of the advantages and limitations of this approach.

5.5.5.2.1 Class TrussElement

Class `TrussElement` extends class `Element` and implements the following interfaces: `ICollaboratable`, `IDisplayable`, `IElement`, `IOneDElement`, `IFEObject`.

Class `TrussElement` only allows for axial forces in a 2D space. This element requires two nodes with two translation degrees of freedom per node. It solves axial forces in any plane that are parallel to the global XY or YZ planes. The method *setNodalDOFs()* determines which system plane contains the element, and activate the nodal degrees of freedom accordingly.

The fundamental attributes are the following:

- The *persistent identifier* of the cross section object

5.5.5.2.2 Class FrameElement

Class `FrameElement` extends class `Element` and implements the following interfaces: `ICollaboratable`, `IDisplayable`, `IElement`, `IOneDElement`, `IDiscreteElement`, `IFEObject`.

This is a plane frame element that calculates bending, shear and axial forces. This element is also a 2D space element. It requires three degrees of freedom per node, two translations and one rotation. The method *setNodelDOFs()* determines which system plane contains the element, and activates the nodal degrees of freedom accordingly.

The `IDiscreteElement` interface allows the user to use internal forces without compromising the result. The fixed-end forces are stored and when the internal forces are computed, the fixed-end forces are subtracted to obtain an exact result.

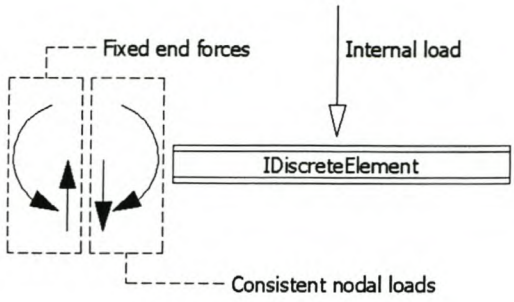


Figure 11 Fixed-end forces of IDiscreteElements

The fundamental attributes are the following:

- The *persistent identifier* of the cross section object
- The rotation angle of the element about its *x-axis*

5.5.5.2.3 Class CST (Constant Strain Triangular element)

Class `CST` extends class `Element` and implements the following interfaces: `ICollaboratable`, `IDisplayable`, `IElement`, `IFEObject`, `ITwoDElement`.

The pilot implementation of this element is done in planes that are parallel to the global XY plane. This element calculates membrane stresses and strains. The in-plane bending performance of the element is poor, and care must be taken to generate a fine mesh.

5.5.5.3 3D space elements

5.5.5.3.1 FrameElement3D

Class `FrameElement3D` extends class Element and implements the following interfaces: `ICollaboratable`, `IElement`, `IFEObject`, `IOneDElement`.

This element is a standard frame element in a 3D space. It has two nodes with six degrees of freedom per node. The element calculates axial, y and z – shear forces, torsion moment, y and z bending moments.

5.5.6 Class Load

Class `Load` is an abstract class that extends class `FEObject` and implements the following interfaces: `ILoad, ICollaboratable.`

Class `Load` forms the basic building block for all load objects.

A load is a vector; it has an intensity and a direction. It is convenient to describe a load as an intensity in a certain direction, and not as two or three load components in the global system directions. The direction vector can be any vector that is convenient for the user e.g. [1,4,0]. When the load vector is computed, the direction vector is normalised.

The fundamental attributes of this class are the following:

- The *persistent identifier* of the component on which the load acts
- The direction vector – a double array
- The intensity vector – a double array

5.5.6.1 Class NodeLoad

Class `NodeLoad` extends class `Load` and implements the `IDisplayable` interface:

The number of degrees of freedom at a node is determined by the type and orientation of the elements connecting to the node. Example: If a node load of 10kN acts in the direction [0,0,1] and the third translation degree of freedom is not active, the structure cannot accommodate the load, and in the analysis the load will simply not exist. This implies that a node load must have the ability to check that the degrees of freedom that are needed to express the load components in the global coordinate system are available. If this is not the case, the complete effect of the load on the structure is not modelled.

5.5.6.2 Class PointLoad

Class `PointLoad` extends class `Load` and implements the following interfaces: `IDisplayable, IDiscreteElementLoad, IPointLoad.`

Class `PointLoad` is used to model internal concentrated loads on DiscreteElements. This class has the ability to calculate the internal forces resulting from the point load. A DiscreteElement has a set of DiscreteElementLoads. When the *getLoadVector()* method of a *DiscreteElementLoad* is invoked, the *DiscreteElementLoad* will return the load vector and add itself to the set of *DiscreteElementLoads* in the *DiscreteElement*. This registration is done to enable

the *DiscreteElement* to find the *DiscreteElementLoad* in order to calculate the internal forces of the *DiscreteElement*.

DiscreteElementLoads also check that they can be fully expressed in the element coordinate system.

5.5.7 Class Support

Class `Support` extends class `FEObject` and implements the following interfaces: `ISupport, IDisplayable, ICollaboratable`.

Class `Support` provides kinematic boundary conditions. In this implementation, there are six possible support conditions: three translations and three rotations. The active degrees of freedom are determined by the node where the support exists. The support conditions only specify that if the nodal degree of freedom is active, then the support condition will apply.

The fundamental attributes are the following:

- The *persistent identifier* of the node where the support exists
- The support conditions (see 0) – a boolean array
- The prescribed displacements – a double array

5.5.8 Class Material

Class `Material` extends class `FEObject` and implements the `IMaterial, IDisplayable` and `ICollaboratable` interfaces. The implementation of `IMaterial` is done for linear elastic materials only. The framework provides facilities for other material types and material matrices.

The fundamental attributes are the following:

- The *Young's modulus* of the material – a double value
- The *Poisson ratio* of the material – a double value

5.5.9 Class CrossSection

Class `CrossSection` extends class `FEObject` and implements the `ICrossSection`, `IDisplayable` and `ICollaboratable` interfaces. This class provides cross sectional properties for 1D elements. Figure 4 shows the coordinate system of a cross section.

The fundamental attributes are the following:

- The *area* of the cross section
- *Ixx* the second moment of the area about the x-x axis
- *Iyy* the second moment of the area about the y-y axis
- *J* the St Venant torsion constant

5.5.10 Class LoadCase

Class `LoadCase` extends class `FEObject` and implements the following interfaces: `ILoadSet`, `IDisplayable` and `ICollaboratable`.

An instance of class `LoadCase` contains a set of `ILoad` objects that act on the structure.

The `analysis` object stores the results of an analysis in the *resultVector* of a load set. Thus, a `LoadCase` instance stores the input of the analysis, (the load vector) and the result of the analysis, (the result vector) together.

The fundamental attributes are the following:

- The *persistent identifiers* of the `ILoad` objects in the load case
- The result vector of the `LoadCase` – a double array

5.5.11 Class LoadCombination

Class `LoadCombination` extends class `FEObject` and implements the `ILoadSet` interface. An instance of class `LoadCombination` combines one or more `LoadCase` objects. It allows the user to combine different `LoadCase` objects using different partial load factors. The `IDisplayable` interface was not implemented in the pilot implementation of this class, thus it is not possible to view `LoadCombination` instances in the Graphical User Interface (GUI).

5.6 Solving the model

The `analysis` object in the Finite Element Application performs a linear elastic analysis on a model. In order to perform an analysis, the references between all the model components in the model are set by the `model` object.

The analysis object creates an `Equation` object that contains the system matrix, the system vector (i.e. the load vector), and the primal and dual vectors.

5.6.1 Steps of the analysis process

The analysis is started by invoking the *perform()* method of class `Analysis`. The following steps occur in the analysis process:

- The set containing the *IDiscreteElements* is traversed and the following methods of the DiscreteElement are invoked:
  - resetFefs() – this method clears the fixed end force vectors
  - resetLoads() – this method clears the set containing discrete element loads
- If the model contains `Constraint` objects, the master indices of the dof objects are set by traversing the *IDof* set.
- The status vector is assembled. This vector contains a true value for all the primal dofs and a false value for all the dual dofs.
- Compute the system stiffness matrix by summation of all the element stiffness matrices, which are obtained by traversing the *IElement* set.
- If the model contains `Constraint` objects, the compatibility **[A]** is now calculated, followed by the computation of the reduced system matrix. See paragraph 5.5.4.1.1.
- The system matrix is decomposed, using Cholesky decomposition, into a lower triangular matrix. [7,8]
- The next step involves a forward and back sweep over the system equations for each load set in the model. The result vector (i.e. the values that are computed) of each load set is stored in the corresponding load set.

Element results are computed on user demand for a specified load set (i.e. the active load set) by back substituting the result vector into the degree of freedom objects and invoking the *getElementResultVector( )* method of an element.

6. Graphical User Interaction

This part of the thesis defines an object-oriented framework for graphical user interaction, which complements the Finite Element framework. The necessary interfaces and classes are developed and implemented. Detailed descriptions of the classes are documented in the Java documentation of the application.

Referring to the criteria for success (section 1.2) for the graphical user interface, much effort was extended to

- make the interface easy to use
- allow new graphical elements to be developed by implementing well-defined interfaces
- allow the user to obtain specific information about specific objects

6.1 The graphical user interaction concept

By enabling the engineer to graphically represent structural concepts, the computer model of the structure can be constructed effectively. The graphical user interface should allow the engineer to obtain specific results from the analysed structure and not flood him with information, e.g. the engineer should be able to get the displacements at one node only, or obtain the axial force in a selected truss element.

The graphical tools that provide the necessary support are discussed in the remainder of this chapter.

Figure 12 shows the layout of the graphical user interface of the pilot application.

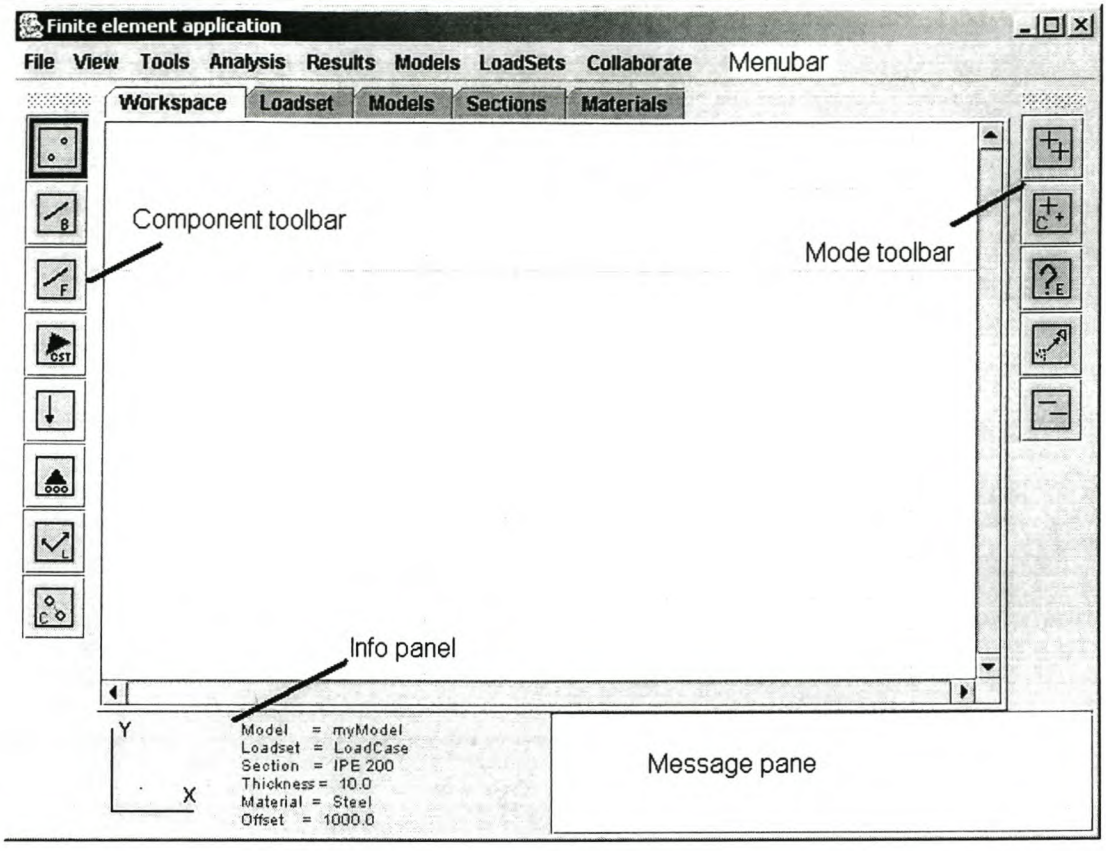


Figure 12 The graphical user interface

6.1.1 Drawing surface

The drawing surface enables the engineer to create the finite element components by drawing them on the drawing surface. This is possible since the required finite element components implement the `IDisplayable` interface. To assist the engineer in the assembling of the structure, a grid is provided, and the functionality to snap to the grid. The size of the grid is prescribed by the engineer, and the grid can be switched on or off.

One of the most important functions provided by the drawing surface is that it allows the engineer to select a certain finite element component by clicking on it with the mouse. This functionality enables the engineer to connect elements to nodes, to move nodes, etc.

6.1.2 Component toolbar

The component toolbar contains the available graphical finite element components (gui components). The gui components displayed in the component toolbar are objects that implement the `IGuiComponent` interface and extend the class `GuiComponent`.

The following *GuiComponents* where implemented in the pilot application:

1. GuiNode
2. GuiTrussElement
3. GuiFrameElement
4. GuiCST (Constant strain triangle)
5. GuiPointLoad
6. GuiSupport
7. GuiLocal
8. GuiConstraint

A `GuiComponent` allows the engineer to create, copy, edit, move and delete finite element components. For example, the `GuiNode` provides the functionality to create, edit, move and delete node objects in the active model.

6.1.3 Mode toolbar

The mode toolbar enables the engineer to select an operating mode. The following modes are available:

1. Add
2. Copy
3. Edit
4. Move
5. Delete

The mode toolbar specifies the mode in which operations are performed: If the add mode and the `GuiNode` button are selected, the user will add new node objects to the active model by clicking the positions of the new nodes. If, for example, the move mode is selected, the user will be able to change the coordinates of a node by dragging it to a new position. By separating the mode and the component toolbars, a matrix of operations is provided where an operation is defined as any combination of a component and mode.

6.1.4 Workspace and tab panels

The workspace provides the drawing surface of the graphical user interface. This is where the user can view the finite element model, and perform operations on the model. Other panels are provided for the user to perform selections of model components that are not graphically displayed in the model, e.g. the existing load sets in the active model. These are:

- Models – This panel displays all the models that are available in the Finite Element Application.
- Loadset – This panel displays all the available load sets in the active model.
- Sections – This panel displays the cross sectional objects in the active model. The cross sections are used by 1D elements.
- Materials – This panel displays all the materials in the active model.

6.1.5 Menu bar

The menu bar provides functionality in the graphical user interface that is not directly related to specific components, e.g. the creation of a new model.

6.1.6 Message to User panel

This panel displays messages from the application to the user. Method **messageToUser(***String s***)**, provided in class `FEObject`, will display a message to the user on this panel.

6.1.7 Info panel

To support effective model production, the info panel displays the following current variables:

- Model – the *persistent identifier* of the active model
- Loadset – the active load set in the current model
- Section – the active section, this is the section that is assigned to 1D elements that are created
- Material – the active material, this is the material that is assigned to all the elements that are created
- Offset – this is the third coordinate of the position of the mouse. See 9.3 for a detail discussion of the variable
- Coordinate system – the global axis of the current view is shown

6.1.8 Popup menus

Interface `IGuiComponent` contains the following method: *popMenu( )*. This method provides the user with options related to the specific object on which a right click of the mouse was performed. This functionality allows the user to interact with specific objects in the model. The primary function of the popup menu is to select the display mode of individual objects.

6.1.9 Getting user input

The process of obtaining information from the user was generalized by the introduction of a generalized data-enter-dialog. This is done by class `DataEnterDialog` that displays an array with `IDialogItems`. The following implementations of interface `IDialogItems` were created.

Table 1 Implementations of interface `IDialogItem`.

DialogCheckEntry	Contains a `JCheckBox` which allows the user to select an entry.
DialogCheckEntry2	Contains a `JCheckBox` and a `JTextField` which allow the user to select an entry and to enter a string value associated with the entry.
DialogComboEntry	Contains a `JComboBox` which allows the user to select an item from a list or set.
DialogComboEntry2	Contains 2x `JComboBox` and a `JTextField` which allow the user to select an item from each of the two lists and to enter a string value associated with the selected items.
DialogLabels	Displays a String array of labels in a `DataEnterDialog`.
DialogTextEntry	Contains a `JTextField` which allows the user to enter string values.

Figure 13 shows some examples of data enter dialogs.

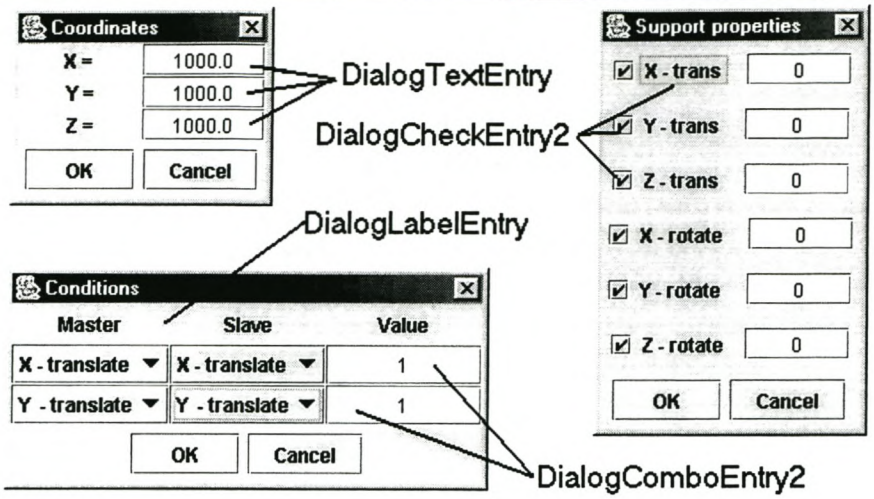


Figure 13 Some examples of DataEnterDialogs

7. Collaboration

The final part of the thesis is aimed at supporting multiple users to collaborate in the execution of a project's analysis tasks. This requires facilities for the distribution and exchange of the project's analysis parameters. The first part of this chapter will discuss collaboration in general. After that, the remainder of the chapter will deal with the implementation of the fundamental collaboration-supporting functionalities in the pilot application.

In the information age, the sharing of information forms the basis of the world economy. Companies with the ability to manage and share information in a useful way will have a better chance of survival.

7.1 Collaboration in Structural Analysis

The nature of structural analysis is such that a large number of small changes are made to the analysis information base. Consequently, the exchange of data files is not adequate to support professional interaction. Furthermore, some analysis tasks can be performed simultaneously, which requires the exchange of information at the level of the objects.

Collaboration is made possible by the object-oriented approach; the data has meaning outside the context of an algorithm, which allows transmission in small units.

Support of collaboration, where a team is working together on a project on the basis of communication supported by a network, is to be provided. The team members can be distributed in time and location. This creates a new set of problems that have to be solved in order to achieve the aims.

7.2 Properties required to enable successful collaboration

The problems which have to be addressed to enable collaboration in the structural analysis environment are discussed below.

7.2.1 Standalone capability

The standalone capability is an aspect that is important for the structural analysis process. This allows an engineer to work when a connection to the network is not available. In addition, it provides a safety mechanism that protects the work of an individual in the communication network. Standalone capability creates the problem of keeping the information consistent between users.

7.2.2 Consistency

Consistency is a problem that surfaces once different sets of the same information exist. Consider, for example, User1 and User2 working simultaneously on the same model. User1 introduces a fundamental change to the model that directly affects the work of User2. User2 must be aware of the change. If that is not the case, User2 is using inconsistent information without knowing it. The use of inconsistent information has a negative effect on the design process.

7.2.3 Flexibility

Care must be taken to ensure that collaboration increase the productivity of the engineering process; it should provide tools that support and enhance the engineering process.

7.2.4 User control over updating of the information base

Due to the iterative nature of structural analysis, the control of storing and updating of information should be with the engineer, not the system. It is not practical to update the information base for every change in a model. That would lead to proliferation of versions that are not useful to the engineering process.

7.2.5 Media – suitability

With the current capacity of communication networks, it is important to transmit small units of information. Care must be taken not to transmit unnecessary information, since this would slow down the collaboration process and frustrate team members.

7.3 Introduction of Collaboration concepts into the pilot implementation

In the paragraphs above, the properties that a system requires to enable collaboration were discussed. The technological basis of the implementation, and aspects that were introduced in the pilot application are described below. See paragraph 4.2.3 for the `ICollaboratable` interface description. This interface must be implemented by all the finite element components that are used in the collaboration process.

7.3.1 Network communication technology

RMI: The pilot application uses the remote method invocation (RMI) technology of Java to transfer information between the Finite Element Applications and the FEAServer (Finite Element Application server). Using RMI, a Finite Element Application can invoke a method on the `FEAServer` and the other way around. To access a remote object, that object's class must extend the java

`UnicastRemoteObject` class, and implement an interface that extends the Java `Remote` interface. The functionality of the said interface will be available to objects that have a reference on the remote object. [5]

7.3.2 Server services

The server: To support collaboration, a server is required to provide certain functionalities. The server must be located on a computer that is able to provide continuous support on the communication network. The capacity of the server must be adequate to perform the tasks that the different users require simultaneously.

Pilot implementation: On the Finite Element Application side, the remote interface `IRemoteWorkplace` defines the functionality that the server requires to enable collaboration. On the side of the server, the interface `IFEAServer` defines the functionality that the distributed Finite Element Applications require from the server. See appendix A for a detailed description of these interfaces.

7.3.3 Identification of objects

Structure of persistent identifiers: The structure of a *persistent identifier* consists of a static part and a dynamic part. The static part is a descriptive string that gives some meaning to the identifier e.g. User1.Node.. The dynamic part is a counter that makes the identifier unique.

Persistent identification service: Paragraph 3.1 discusses the importance of unique identifiers for objects existing in an application. To enable collaboration, the *persistent identifiers* must be unique in the entire project space. This is done by a *persistent identifier service* that the FEAServer provides. When a Finite Element Application creates a finite element component, the application will request a *persistent identifier* from the FEAServer. To streamline the process, the FEApplication requests a batch of *pids* at a time, and stores them locally. If the local buffer level of *pids* is low, the FEApplication will request additional *pids* from the FEAServer. The persistent identifier service contains a HashMap that uses the static part of an *persistent identifier* to map an integer that gives the number of the last *pid* issued by the service with the particular static part. The *getPID( )* remote method requires a string, i.e. the header part of the *persistent identifier* and an integer, the number of *persistent identifiers* required, as parameters. This method returns a `batch` of *persistent identifiers.*

7.3.4 Perception of fundamental changes

Any `ICollaboratable` object must know when it undergoes a fundamental change. This is extremely important to the following services that support collaboration.

- Versioning
- Consistency

In the pilot implementation, the attribute *fundamentallyChanged* indicates that the object changed since if was last versioned.

7.3.5 Notification of changes

UpdateGenes: The notification of changes forms an essential part of the collaboration environment, especially when standalone capability is needed. When a fundamental change occurs, an `UpdateGene` is created. The `UpdateGene` encapsulates the *selection identifier* of the component that was fundamentally changed, the new values introduced by the change, the name of the user who performed the change and the name of the method that caused the change.

Pilot application: The `UpdateGene` is send to the FEAServer for processing. In the pilot implementation, the server distributes the `UpdateGenes` to all the users that are logged into the FEAServer at the time that the fundamental change occurs. In a commercial application, the server will only send the `UpdateGene` to users that are using the same version of the object in consistent mode (See paragraph 7.3.9).

7.3.6 Updating of changes

The FEAServer distributes `UpdateGene` objects to a remote workplace by the invocating the remote method: *notifyFundamentalChange ( )* at a remote workplace. The `UpdateGene` is send to the remote workplace as a parameter of the *notifyFundamentalChange( )* method.

In the pilot implementation, the *notifyFundamentalChange( )* method automatically updates the finite element component that matches the *selection identifier* contained in the `UpdateGene`.

In a commercial application, the user must control the updating process because:

- The user must be aware of a fundamental change that occurs in his model
- The user must accept or reject an update using own judgement.

7.3.7 Versioning

User control: Versioning is an important issue that arises in a collaborative environment. As mentioned in paragraph 7.2.4 the engineer must be in control of the versioning process of a model to avoid a proliferation of versions. However, this implies that changes have to be accumulated.

Versioning service: By making use of the *versioning service* provided by the FEAServer, a new `Version` object is created for all the finite element components that were fundamentally changed (since the last version) at a time the user determines. The versioning service in the pilot application makes use of a HashMap that maps a *persistent identifier* to an integer value. By invoking the *getNewVersionNumber()* method, the latest version number of a specific *persistent identifier* is obtained.

7.3.8 Transmitting objects over a communication network

Streamed format generation: It is important that the `Collaboratable` objects can generate a compact streamed format. When an object is sent over a network, the object is written into a byte stream. To ensure that only the fundamental attributes are written into the byte stream, all the `Collaboratable` objects implement the `Externalizable` interface of Java. This interface requires the following methods:

- *writeExternal()*
- *readExternal()*

In the pilot application, only the fundamental attributes are written to the output stream. Related objects are not automatically written to the output stream. This ensures that the minimum amount of data is transmitted over the network.

Pilot implementation: The pilot implementation allows one user to send a set of `Collaboratable` objects to another user to demonstrate the transmission of objects over a communication network. Once the objects arrive at the receiving end, they are added to the active model, and referenced in the application map. Fundamental changes to these objects are registered at the FEAServer, and distributed to other users (see paragraph 7.3.5 and 7.3.6).

7.3.9 Storage of components

In a commercial application, a *central database service* is required to store the finite element components. Finite element components can then be checked out of the server. Two modes of checking out of objects are proposed:

- Consistent mode – If an object is checked out in consistent mode, fundamental changes are distributed to users that use the same version of the object in consistent mode.
- Free mode – If an object is checked out in free mode, no change reports are created when the object changes.

The storing of finite element components is outside the scope of this thesis.

7.4 Geometrical description

The information required about a structure reaches beyond the analysis of the structure only. An analysis model is based on the geometrical description of the structure. If the structure and semantics of the geometrical information is known, it is possible to map the geometrical parameters to the analysis parameters. However, a proper mapping of the geometrical parameters to an analysis model requires a level of engineering experience that cannot be provided by software alone. Tools must be provided to assist the engineer in doing this mapping.

Displaying the geometry of a structure in the analysis workspace (see paragraph 6.1.1) enables the engineer to easily create an analysis model that complies with the geometrical properties of the structure. This gives the engineer full control over the mapping process and the analysis. The engineering is done by an engineer not a computer. The computer supports the engineer with powerful tools.

8. Pilot Application Examples

This chapter describes the working of the pilot implementation by means of a few examples.

8.1 Truss analysis

The analysis of a truss is shown. The truss spans 4.0 m and is 0.8 m deep. All the cross sections are 60x60x4L and three 60kN loads act on the truss.

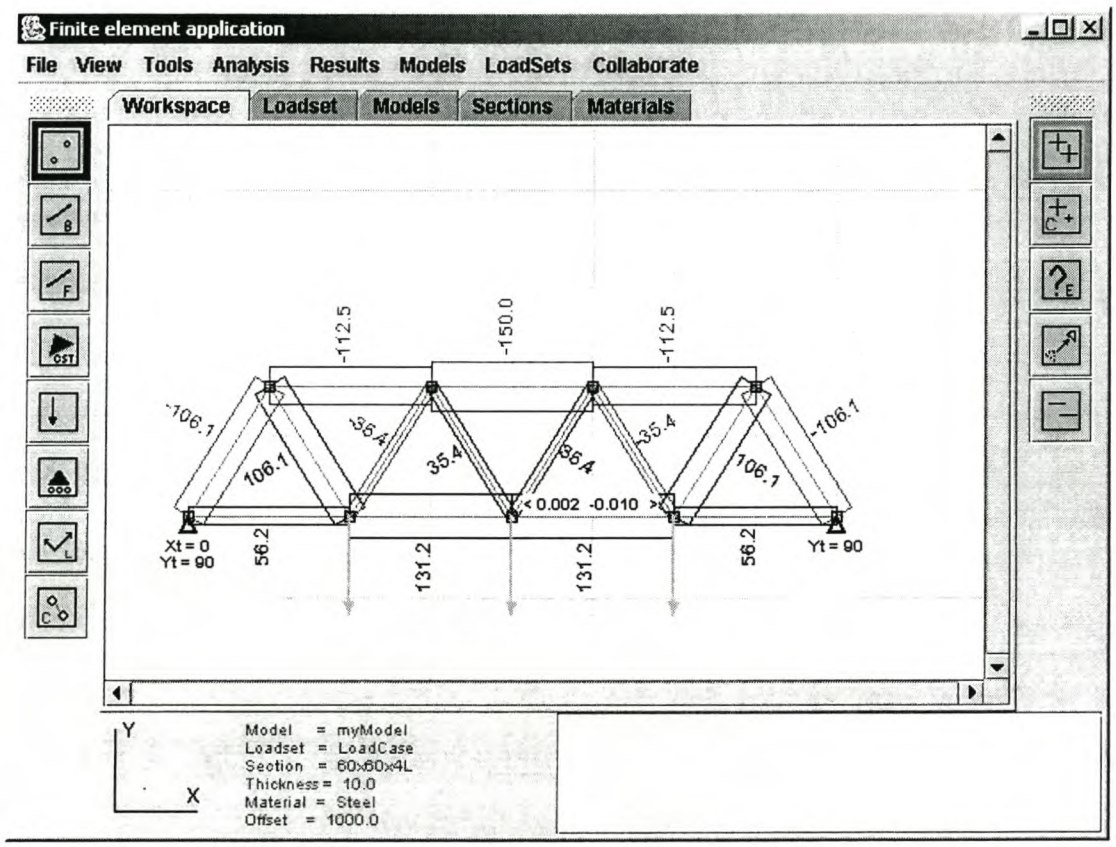


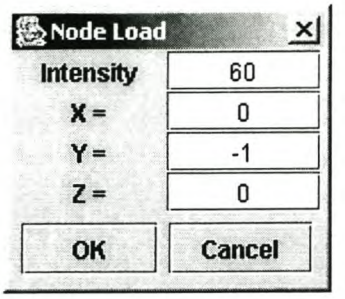
Figure 14 Truss analysis example

Steps in creating the model

- Create the *nodes* by clicking on the grid positions where the nodes are. (Coordinates can also be set by selecting the *edit* mode, and clicking of an existing node. A popup dialog will appear which allows the user to enter the coordinates numerically)
- Select the *sections* tab, and click on the 60x60x4L object. This selects the cross section for the truss elements.

- Select the truss element mode by clicking the button. Draw the truss elements. This is done by selecting the node where the first truss element starts, the second node that is clicked indicates the end point of the first truss element and the starting point of the second truss element. To select a different starting point as the ending point of the last truss element, click on the button and select the new starting point, followed by the endpoint.

- Select the button to add the node loads. Click on the nodes that have the node loads. A popup dialog will appear which asks the user the intensity as well as the vector components of the direction vector of the load. In this example, the node loads are 60kN each.



- Select the button. Add the supports, on the left hand side X and Y translations and on the right hand side Y translation.
- Analyse the model by selecting the *analyse model* command from the analysis menu.
- Select AFD (i.e. axial force diagram) from the results menu.
- By selecting the node mode button and right clicking on the middle node, the displacements of the node can by viewed. (Select *displacements* from the popup menu.)
- By selecting *reactions* from the popup menu on the supports, the reactions of the analysis can be viewed. (Note: to activate the support popup menu, the support mode must be selected, and the mouse must be right clicked on a support)

8.2 Combination of CST and frame elements

This example shows how to connect a frame element to constant strain triangular (CST) elements using constraints.

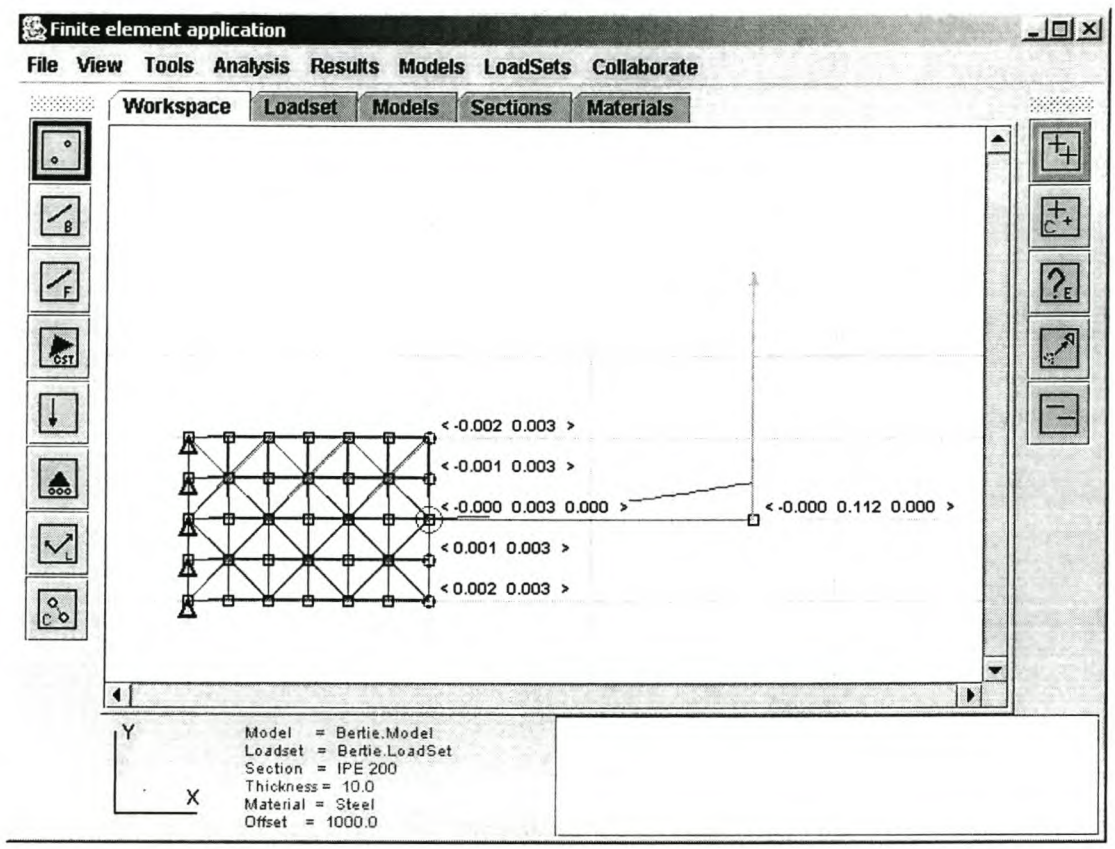
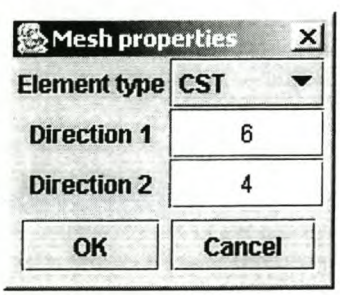


Figure 15 Frame element combined with CST element

Steps in creating the model

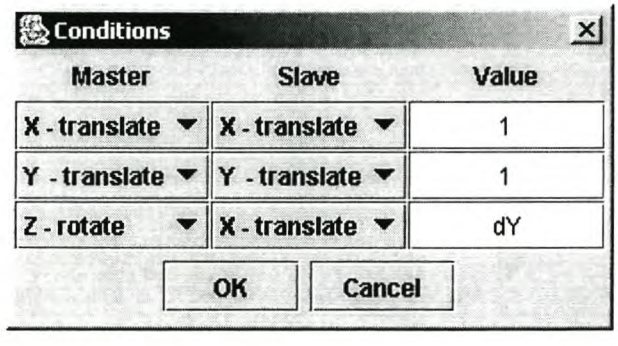
- Create the four corner nodes of the area to be meshed using CST elements.
- Create the mesh by using the *mesher* option. (Select mesher from the Tools menu). The directions are defined as follows: Direction 1 is the direction from the first node clicked to the second node that is clicked. Direction 2 is the direction from the second to the third node that is clicked. The numbers in the text fields next to the directions indicates the number of required elements in each direction. In this example, six elements are required in direction 1 and 4 elements in direction 2.



- Add the supports by selecting the support button and clicking on the top left node. Select the X and Y translation to be supported. The remaining supports

can be added by selecting the copy mode, . Select the existing support (i.e. click on the existing support) and then select the nodes where equivalent supports are required. Equivalent supports to the existing support are created at these nodes (i.e. supports with the same conditions)

- Create the node on the right hand side that forms the right end of the frame element.
- Create the frame element (Using an IPE 200), by selecting the start and end nodes that define the element.
- Add the constraint conditions by selecting the constraint mode, .
    - Select the master node
    - Select the slave node
    - Specify the number of conditions between the master and the slave node, three in this example.
- The following dialog appears which allows the user to specify the constraint conditions. In the example the beam must be connected to the CST mesh in such a way that the full moment at the left hand side of the beam is transferred into the CST mesh. The slave nodes are defined as the nodes that connect to the frame element, thus all the nodes on the right hand side of the CST mesh are slave nodes (except the master node, i.e. the node to which the frame element connects). Thus, the X-translation of the beam's end node must be connected to the X-translations of the CST mesh slave nodes. The Y-translation of the beam must be connected to the Y-translations of the slave nodes. The Z-rotation of the beam at the left hand side must be connected to the X-translation of the slave nodes multiplied by the lever arm to the slave nodes. Note that the value **dY** in the third text field is equal to the y-coordinate of the master node minus the y-coordinate of the slave node. The numerical value is calculated during the analysis of the model. For a positive rotation at the left hand side of the beam, the top node will translate in the negative X-direction and the bottom node will translate in the positive X-direction. After the first constraint condition is created, the other conditions are created by using the copy mode. This is done



by selecting the copy mode, selecting the constraint, and selecting the new slave nodes.

- Adding the load; in this example, a load of 150kN is added in the direction [0,1,0] to the right hand side of the beam element.
- Analyse the model by selecting the *analyse model* option from the analysis menu.
- By showing the displacements of the master and slave nodes, it can be clearly seen that the constraint conditions are met, the Y-translations are the same, and the X-translations increase linearly from the master node.

8.3 Cantilever beam with point loads

This example shows a cantilever beam with two concentrated loads and a local coordinate system at the free end.

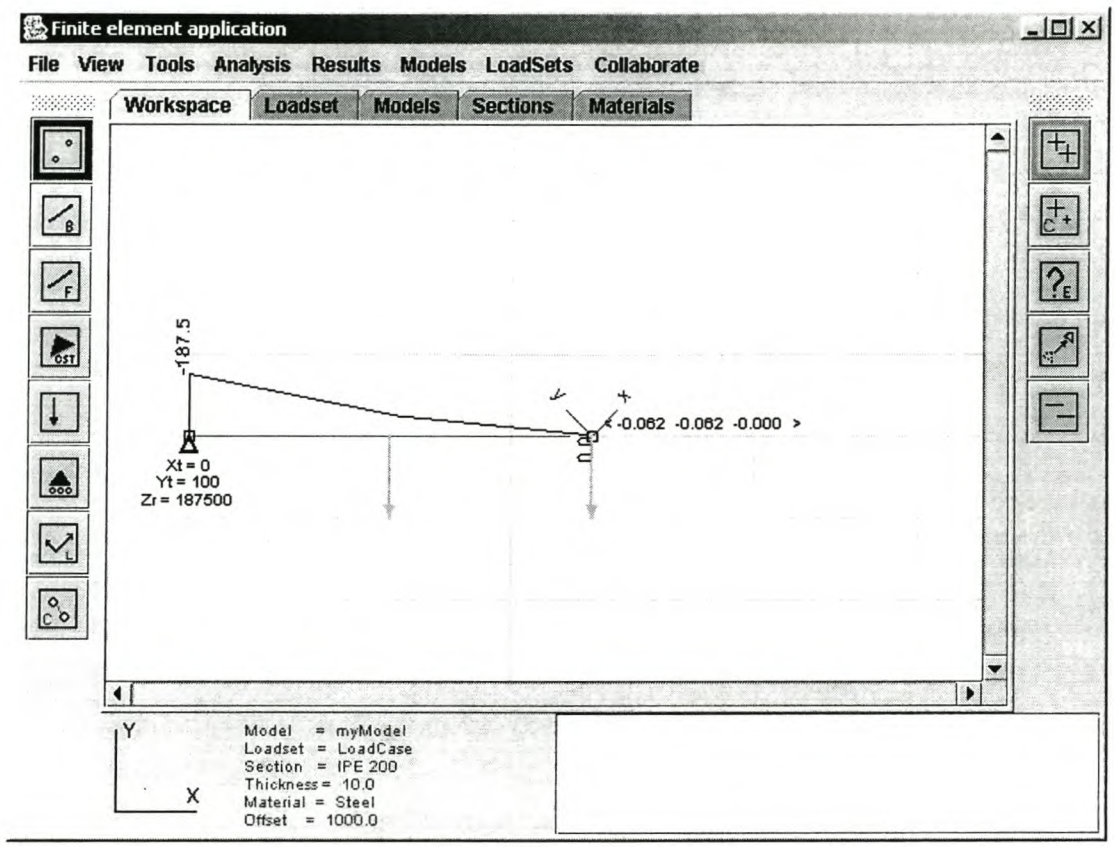
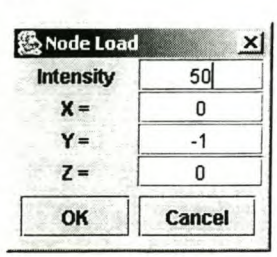


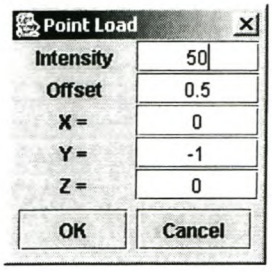
Figure 16 Cantilever beam with concentrated Loads

Steps in creating the model:

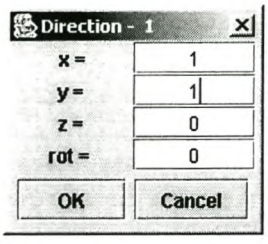
- Create the nodes by clicking the node positions.
- Create the plane frame element by selecting the button and clicking on the start and end nodes.
- Create the support by selecting the button and clicking on the left node. Select the following support conditions: X-translate, Y-translate and Z-rotate.
- Select the button to add the concentrated loads.
    - Click on the right hand side load. A node load dialog appears which allows the user to enter the intensity and direction vector components of the node load.



- Click on the frame elements. A point load dialog appears which allows the user to enter the intensity, offset from the first node and the direction vector components of the load



- Select the button and add a local coordinate system at the node on the right hand side. A dialog appears that allows the user to enter the direction vector (i.e. the direction of the x-axis of the local coordinate system) and the rotation angle of the coordinate system about this axis.



- Analyse the model by selecting the *analyse model* command from the analysis menu
- View the bending model diagram by selecting *BMD* from the results menu

Figure 16 shows the bending moment diagram of the beam, the reactions and the displacement of the free end. The section is an IPE 200, with a point load of 50kN at mid span and a 50kN load at the free end. The local coordinate system makes a 45° angle with the global coordinate system.

8.4 Using 2D frame elements in different global planes

This example shows the use of two identical frame elements together in different global planes. Figure 17 shows a top view on the model. A node load of 100kN is added at each of the free ends. (The node loads are running into the screen, and are not clearly visible on the figure). At the bottom left node the Y-and Z- translations and the X-rotation are active. (See Table 2) At the top right node, the active degrees of freedom are the X- and Y-translations and the Z-rotation. At the top left node all the degrees of freedom are active except the Y-rotation degree of freedom.

The Y-translations at the free ends of the beams are equal to –0.029 and the Y-reaction is equal to 200kN, the sum of the two node loads at the free ends. The reaction moments are equal in size and opposite in sign (using the right hand rule to determine the signs of the reaction moments)

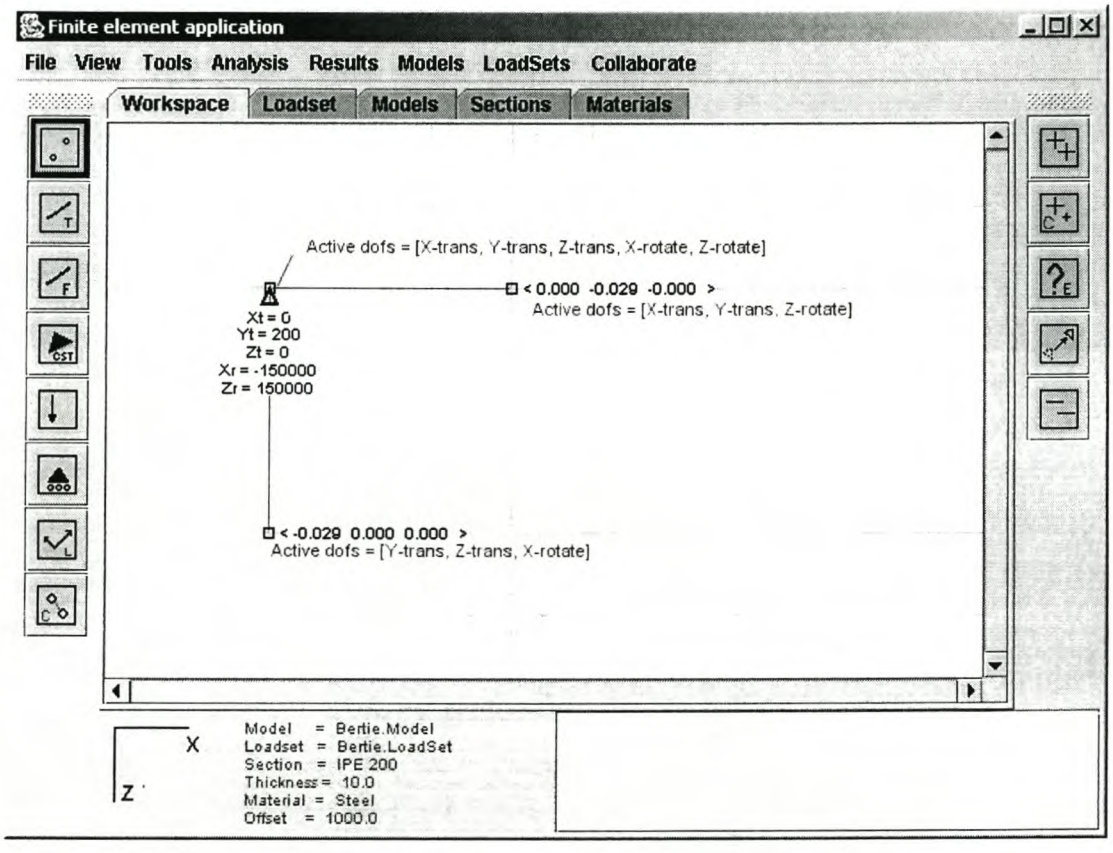


Figure 17 Top view of Frame Elements in different global planes

See section 9.1 for a detailed explanation of the importance of using 2D space elements in a three dimensional space.

8.5 Collaboration

In this example support for collaboration as implemented in the pilot application is demonstrated.

Steps of the example

- Start the *RMI registry* by running the start registry batch file.
- Start the *FEAServer* by running the *startserver* batch file.
- Start the FEApplications by running one or more of the *startclient* batch files.
- Each workplace logs onto the *FEAServer* by selecting the *open connection to FEAServer* from the Collaboration menu.
- Users now have the opportunity to creating different finite element components and to send them to one another. Select the *send* option from the collaboration menu to send a set of components to a different user. A dialog appears that asks for the name of the user and the name of the set to be sent. Once this information is entered, the workplace gets a reference from the *FEAServer* to the remote object of the other workplace that must receive the set of finite element component. Using this reference, the set of components is send directly to the receiving user by invoking the *uploadFEObject()* method at the receiving workplace.
    - When sending *ILoad* components across the communication network, the *ILoad* objects will be added to the active model on the receiving end, but these objects must also be added to a load set in order to be displayed. Thus, the load set that contains the *ILoad* objects must also be send to the other user.
- When a fundamental change occurs to a finite element component, the *FEAServer* is notified and the notification is sent to all the different workplaces (except the workplace that caused the change). When a workplace receives a change notification, it will check whether it contains the finite element component, and will automatically update the change if it does.

The aim of the collaboration part of the pilot implementation is to show that the finite element components:

- can be successfully transmitted across a communication network
- have the ability to know when a fundamental changed occurred
- have the ability to perform updates on themselves

9. Solutions to special problems

9.1 Connecting element degrees of freedom

This problem will be explained by considering a simple example: Say the user wants to use plane frame elements together with plane truss elements. A plane frame element is an element with three degrees of freedom per node, while a plane truss element only has two degrees of freedom per node.

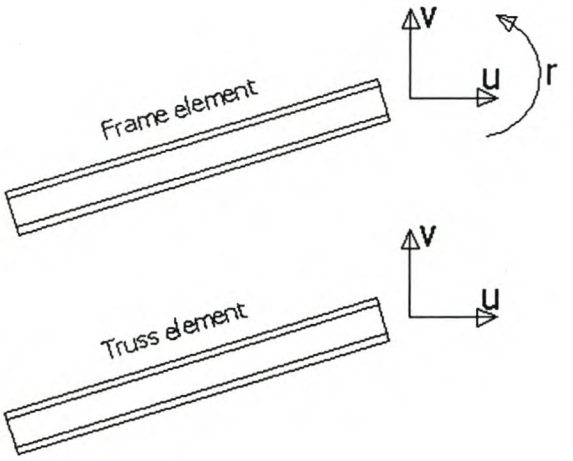


Figure 18 Element degrees of freedom

One way of solving this problem is to allow a node to have any number of degrees of freedom. (There is no limit on the number of degrees of freedom per node, depending on what must be calculated. In three dimensional space, using displacement-based elements, six degrees of freedom are required.) For the pilot application, it was decided to use three translation degrees of freedom per node and three rotation degrees of freedom. Thus, a node will have a maximum of six dofs in the global coordinate system. Table 2 maps the index number of the dof array to the physical meaning of the degree of freedom in the global coordinate system.

Table 2: System degrees of freedom and their physical meaning.

Index	*Physical Meaning*
0	X - translation
1	Y - translation
2	Z - translation
3	X - rotation
4	Y - rotation
5	Z - rotation

All the elements implement a method: *setNodalDOFs( )*. This method will determine which system degrees of freedom to activate at each node to describe the element displacements. If a dof is not needed, it will not be activated, thus there is no need to add boundary conditions to remove dofs with zero stiffness from the system.

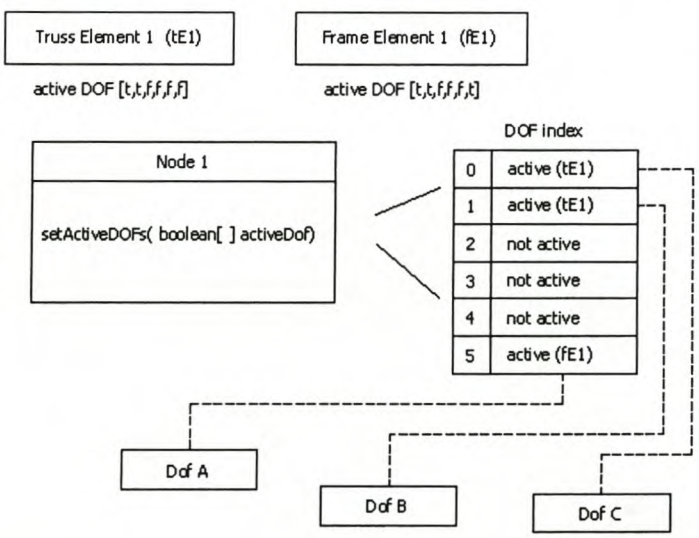


Figure 19 Activating degrees of freedom

In Figure 19 a graphical representation of the nodal degrees of freedom is shown. Firstly the setNodalDOFs(boolean[ ] activeDOF) method of element *tE1* is invoked. This method determines the required system dofs needed to describe the element displacements of element *tE1*, and invokes the setActiveDOFs(boolean[ ] activeDOF) method of *Node 1*. The boolean array *activeDOF* indicates the global dofs that influence the state of element *tE1* during the remainder of analysis process. *Node 1* checks that the dofs element *tE1* require are active. If they are not active, *Node 1* will create a dof object for each dof required. Next, the setNodalDOFs(boolean[ ]

activeDOF) method of element *fE1* is invoked. Element *fE1* in turn invokes the setActiveDOFs(boolean[ ] activeDOF) method of *Node 1,* and *Node 1* checks that the required dofs are active. In this case, *Node 1* activates an additional dof for element *eF1*, namely dof[5] will be activated.

By traversing the entire element set in the model, all the dofs needed to describe the model's state will be created. Conversely, a dof will only be created if it is required by one or more elements.

A significant advantage of this approach is that it allows a user to use 2D space elements in a 3D model. However, their use is limited to the case where the elements are oriented in such a way that the plane in which the element is located has a global axis perpendicular to it or if local coordinate systems are introduced to achieve the above requirement. The element dofs can then be mapped directly to global degrees of freedom, without the possibility of introducing singular dofs.

This approach allows a user to mix different domains (e.g. plane frame, plane truss or space frame etc.) in a specific analysis. Thus, a space frame can by combined with a plane truss analysis without any complications. The user is not limited to a specific domain like in traditional analysis applications.

Example: Say an engineer needs to analyse an industrial steel structure with trusses in the one direction and girders in the direction perpendicular on the trusses. The engineer can model the structure with simple 2D truss elements, because the elements are always perpendicular to one of the global coordinate system directions. This implies that the model consists of 2D trusses arranged in a 3D space. No hinges are required to eliminate unwanted bending moments. Only two equations per node need to be solved except where trusses and girders connect. (See paragraph 8.4. for an example)

9.2 Sparse Matrices

A sparse matrix is a matrix that stores only non-zero values.

Instances of class `SparseMatrix` contain the following attributes:

- A hashMap that maps all the matrix entries. The key for this mapping is a combination of the row and column index. Thus, the key is determined by the position of the `MatrixEntry` in the sparse matrix. This allows for the retrieval of any `MatrixEntry` as a function of the row and column indices.
- An array of matrix entries – the row array. This array references the first `MatrixEntry` in each row. A null reference indicates an empty row.
- An array of matrix entries – the column array. This array references the first `MatrixEntry` in the column. A null reference indicates an empty column.

The building blocks of a sparse matrix are the matrix entries: A matrix entry has the following attributes:

- String *key* – combination of the row and column index as a `String`
- double *value* – the value of the `MatrixEntry`
- row index – the row index of the `MatrixEntry`
- column index – the column index of the `MatrixEntry`
- reference to the next `MatrixEntry` in the row
- reference to the next `MatrixEntry` entry in the column

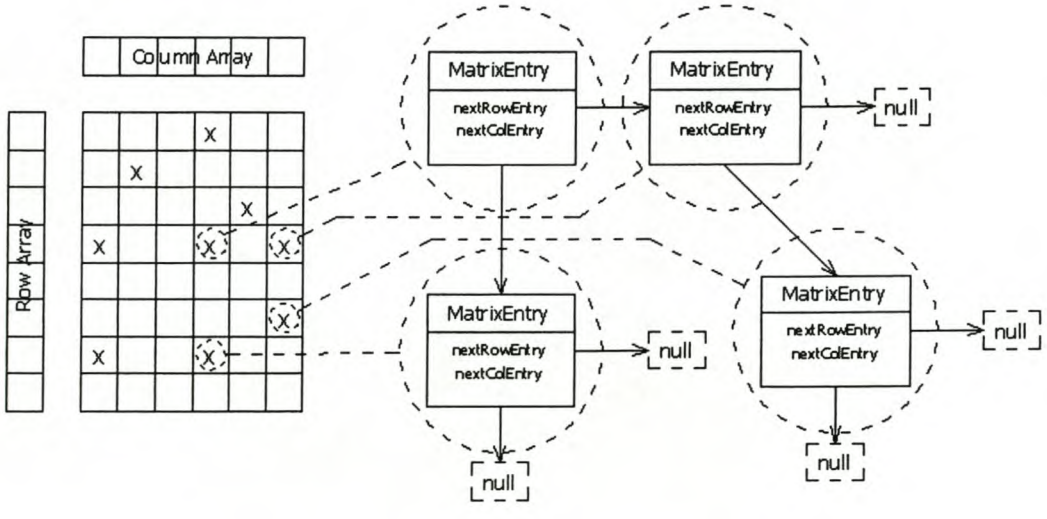


Figure 20 Sparse Matrix configuration

9.2.1 Adding values to a SparseMatrix

Adding a `MatrixEntry` to the `SparseMatrix` involves the following:

1. The entry is added to the entry map (HashMap) of the sparse matrix instance. This creates the possibility to retrieve the entry if the row and column indices are known.
2. The entry is inserted into the row list matching the row index of the entry.
3. The entry is inserted into the column list matching the column index of the entry.

Steps 2 and 3 establish the links between a `MatrixEntry`, and the next `MatrixEntry` in the same row and column. This special linked list allows for fast and effective traversing of the matrix columns and rows.

9.2.2 Retrieving values from a SparseMatrix

Retrieving a value from the `SparseMatrix` is done by invoking the *getValue(int row, int col)* method. Using the values *row* and *col*, the mapping key of the `MatrixEntry` is constructed and the *get(Object key)* method of the entryMap is invoked. If a `MatrixEntry` exists for these indices the value is retrieved, otherwise the value is zero by definition.

9.2.3 SparseMatrix multiplication

Matrix multiplication between two `SparseMatrix` objects is made possible by the twin linked lists of the matrix entries. Figure 21 shows two `SparseMatrices` that are multiplied. Computing $\texttt{MatrixEntry}_{ij}$ in the RESULT_MAT is done by stepping through the i_{th} row of MAT_1 and the j_{th} column of MAT_2. Multiplication is performed on the follow bases:

- If the column index of the `MatrixEntry` in MAT_1 is equal to the row index in MAT_2, the values of the entries are multiplied and stored in a temporary variable.
- If the column index of the `MatrixEntry` in MAT_1 is smaller that the row index of the `MatrixEntry` in MAT_2, the next `MatrixEntry` in this row of MAT_1 is obtained and evaluated.
- If the row index of the `MatrixEntry` in MAT_2 is smaller that the column index of the `MatrixEntry` in MAT_1, the next `MatrixEntry` in this column of MAT_2 is obtained and evaluated.

- When either the end of the particular row in MAT_1 or the end of the column in MAT_2 is reached, the loop terminates, and the temporary value is stored in the RESULT_MAT.

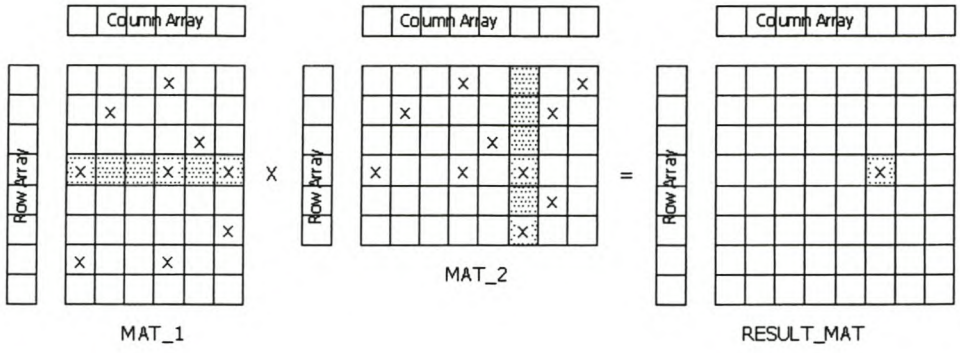


Figure 21 Sparse Matrix Multiplication

Figure 22 shows the inner loop of the matrix multiplication method. This loop is executed for each entry in the result matrix.

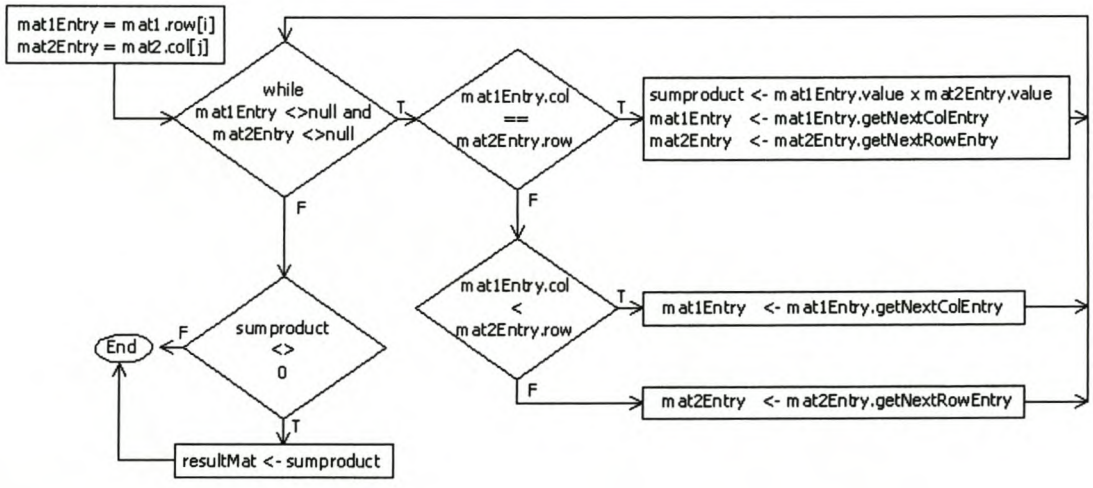


Figure 22 Inner loop of Sparse Matrix Multiplication

9.3 Working in three dimensions on a two dimensional drawing surface

Working in three dimensions causes the problem of determining three coordinates from a two-dimensional working surface, i.e. the computer screen. This problem is addressed by limiting the user to working in 2D planes.

In the pilot application, the plane in which the user works must be parallel to one of the global planes. The offset of this plane to the origin of the global coordinate system is used as the third coordinate of a mouse click and is set by the user.

10. Conclusions

The criteria for success defined in the first chapter were met in this thesis.

The Finite Element method was successfully mapped to the computer. The modular structure of the framework provides a framework that works. The framework can easily be extended.

A graphical user interface that complements the Finite Element framework was implemented with success. This gui allows operations on Finite Element components.

Fundamental collaboration facilities were discussed and successfully implemented in the pilot application.

In Chapter 9 elegant solutions to specific problems that were addressed are discussed. These were:

- The problem of connecting element degrees of freedom
- The problem of using sparsely populated matrices
- The problem of working in three dimensions on a two dimensional drawing surface

References

1. Cook, Malkus, Plesha, *Concepts and Applications of Finite Element Analysis third edition,* Wiley, New York, 1989.
2. West, *Fundamentals of Structural Analysis,* Wiley, New York, 1993
3. Mainçon, *Continuum mechanics and the Finite Element Method,* Lecture Notes at University of Stellenbosch, 2001
4. Zukowski, *Mastering Java 2 J2SE 1.4*, Sybex, 2002
5. Mahmoud, *Distributed Programming with Java,* Manning, 2000
6. Pahl, *Java,* Technische Universität Berlin, 2002
7. Pahl, *Finite Element Theory and Computer Implementation*, Technische Universität Berlin, 2001
8. Pahl, *Finite Element Theory, Stationary Heat Flow*, Technische Universität Berlin, 2000

Appendix A Java doc's

(Content on attached CD)

Appendix B Java code

(Content on attached CD)